Analyse et détection des patrons architecturaux dans les applications mobiles à l'aide des techniques d'apprentissage automatique

par

Chaima CHEKHABA

MÉMOIRE PRÉSENTÉ À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
COMME EXIGENCE PARTIELLE À L'OBTENTION DE LA MAÎTRISE
AVEC MÉMOIRE EN GÉNIE LOGICIEL
M. Sc. A.

MONTRÉAL, LE 1 MAY 2023

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

Chaima Chekhaba, 2023

Cette licence Creative Commons signifie qu'il est permis de diffuser, d'imprimer ou de sauvegarder sur un autre support une partie ou la totalité de cette oeuvre à condition de mentionner l'auteur, que ces utilisations soient faites à des fins non commerciales et que le contenu de l'oeuvre n'ait pas été modifié.

PRÉSENTATION DU JURY

CE MÉMOIRE A ÉTÉ ÉVALUÉ

PAR UN JURY COMPOSÉ DE:

Mme. Naouel MOHA, directrice de mémoire
Département de génie logiciel et des TI

Mme. Ghizlane EL BOUSSAIDI, codirectrice
Département de génie logiciel et des TI

M. Ali Ouni, président du jury
Département de génie logiciel et des TI

M. Sègla Jean-Luc Kpodjedo, membre du jury
Département de génie logiciel et des TI

IL A FAIT L'OBJET D'UNE SOUTENANCE DEVANT JURY ET PUBLIC

LE 18 AVRIL 2023

À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

REMERCIEMENTS

Je tiens tout d'abord à exprimer mes sincères remerciements à tous ceux qui ont contribué à la réalisation de ce mémoire. Je suis très reconnaissante pour leur soutien, leur encouragement et leur confiance tout au long de ce travail.

Je tiens à remercier mes directrices de recherche, Naouel Moha et Ghizlane El Boussaidi, pour leur encadrement, leurs conseils, leur patience et leur dévouement tout au long de ce travail. Votre contribution a été cruciale pour la réussite de ce projet.

Je souhaite également remercier les membres de jurys, Ali Ouni et Sègla Jean-Luc Kpodjedo, pour leur temps et leur expertise. Vos commentaires et suggestions constructives ont grandement aidé à améliorer la qualité de ce travail.

Enfin, je remercie vivement mes parents et ma famille pour leur soutien inconditionnel, leur amour et leur encouragement tout au long de ma vie académique et professionnelle. Votre soutien m'a donné la force et la confiance nécessaires pour persévérer dans les moments difficiles et je vous en suis éternellement reconnaissante.

Analyse et détection des patrons architecturaux dans les applications mobiles à l'aide des techniques d'apprentissage automatique

Chaima CHEKHABA

RÉSUMÉ

Récemment, les applications mobiles ont connu une grande émergence dans de multiples domaines. Afin de répondre à la qualité requise de ces applications, les développeurs mobiles appliquent les bonnes pratiques (par exemple, les patrons) et les directives recommandées par la communauté du développement. Parmi les patrons architecturaux les plus couramment utilisés pour implémenter les applications Android, figurent le célèbre patron architectural Modèle-Vue-Contrôleur (MVC) et ses variantes : Modèle-Vue-Présentateur (MVP) et Modèle-Vue-Vue-Modèle (MVVM). Cependant, les développeurs ont tendance à appliquer différemment ces patrons en fonction de leur expérience et de leur propre compréhension des directives relatives aux patrons. Cela peut conduire à des implémentations incorrectes qui ont un impact négatif sur la conception de l'application. Parfois, cela conduit également à mélanger différentes variantes MVC au sein d'une même application, ce qui peut nuire à la compréhensibilité, à la maintenabilité et à la testabilité de l'application.

Dans ce mémoire, nous nous intéressons à l'analyse des applications Android afin de détecter les patrons appliqués et identifier potentiellement le mélange de plusieurs patrons de type MVC implémentés dans une même application. À cette fin, nous présentons nos deux approches outillées : Coach et MlCoach. L'approche Coach permet de détecter le patron du type MVC dominant dans les applications Android à l'aide de la classification traditionnelle. Quant à l'approche MlCoach, elle permet de détecter le mélange de plusieurs patrons de type MVC à l'aide de la classification multi-label en utilisant l'apprentissage profond. En l'absence de données d'entraînement, nous effectuons nos analyses sur un ensemble de données annotées manuellement de 69 applications Android libres écrites en Java. D'après nos expérimentations, nos approches donnent des résultats prometteurs et surpassent les approches de l'état de l'art. Nous présentons également une analyse qualitative sur un sous-ensemble d'applications mobiles afin de donner un aperçu sur les facteurs qui peuvent conduire au mélange de différents patrons de type MVC dans ces applications.

Mots-clés: patrons architecturaux, MVC, MVP, MVVM, applications mobiles, classification, classification multi-label

Architectural Pattern Detection and Analysis in Mobile Apps Using Machine Learning Techniques

Chaima CHEKHABA

ABSTRACT

Recently, mobile applications (apps) have known widespread emergence in multiple fields. In order to meet the required quality for these apps, mobile developers apply good practices (e.g., patterns) and guidelines recommended by the development community. Among the most commonly used architectural patterns to implement Android apps are the popular Model-View-Controller (MVC) architectural pattern and its variants : Model-View-Presenter (MVP) and Model-View-ViewModel (MVVM). However, developers tend to apply differently these patterns according to their experience and proper understanding of the patterns guidelines. This may lead to faulty implementations that impact negatively the design of the app. Sometimes, it also leads to the mixing of different MVC variants within the same app, which could hinder the understandability, maintainability, and testability of the app.

In this thesis, we are interested in analyzing Android apps in order to detect the applied patterns and potentially the mixing of multiple MVC variants implemented in the same app. For that end, we introduce our tool-based approaches : Coach and MlCoach. Coach allow the detection of the dominant MVC pattern variant in Android apps using the traditional classification techniques. As for MlCoach, it detect the mixing of multiple MVC pattern variants using the multi-label classification technique based on deep learning. In the absence of benchmarks, we perform our analyses on a manually annotated dataset of 69 open-source Java projects for Android apps. Based on our experiments, our approaches yield promising results and outperform the state-of-the-art approaches. We also present a qualitative analysis on a subset of mobile apps to provide some insights about the factors that may lead to the mixing of different MVC variants in these apps.

Keywords: Architectural pattern, mobile apps, MVC, MVP, MVVM, classification, multi-label classification

TABLE DES MATIÈRES

Page

CHAPITRE 1 INTRODUCTION 1
1.1 Contexte et problématique 1
1.2 Objectifs de la recherche 4
1.3 Contributions 4
1.4 Publications 5
1.5 Paquet de réplication 5
1.6 Organisation du mémoire 6

CHAPITRE 2 CONCEPTS DE BASE 7
2.1 Plateforme Android 7
2.1.1 Architecture et principaux composants 8
2.2 Patrons architecturaux 13
2.2.1 Modèle Vue Contrôleur 15
2.2.2 Modèle Vue Présentateur 16
2.2.3 Modèle Vue Vue-Modèle 17
2.2.4 Implémentation des patrons de type MVC en Android 18
2.3 Conclusion 21

CHAPITRE 3 REVUE DE LITTÉRATURE 23
3.1 Analyse d'architectures et de patrons architecturaux dans les applications mobiles 23
3.2 Détection automatique de patrons architecturaux dans les applications mobiles 25
3.3 Détection automatique d'anti-patrons et défauts de code dans les applications mobiles 27
3.4 Utilisation de l'apprentissage automatique pour la détection de patrons, d'anti-patrons et de défauts de code 28
3.4.1 Détection de patrons basée sur l'apprentissage automatique 28
3.4.2 Détection d'anti-patrons et défauts de code basée sur l'apprentissage automatique 30
3.5 Synthèse générale 32

CHAPITRE 4 DÉTECTION DES PATRONS DE TYPE MVC DANS LES APPLICATIONS ANDROID BASÉE SUR LA CLASSIFICATION : L'APPROCHE COACH 35
4.1 Prétraitement 35
4.1.1 Extraction des classes significatives 37
4.1.1.1 Classes modèles (CMs) 37
4.1.1.2 Classes d'interaction avec l'utilisateur (CIUs) 38
4.1.1.3 Classes de liaison de données (CLDs) 38

4.1.1.4 Classes vue-modèles (CVMs) 38
4.1.1.5 Classes intermédiaires (CIs) 39
4.1.2 Calcul des métriques de code 39
4.1.2.1 Vecteur de caractéristiques de type CIU 40
4.1.2.2 Vecteur de caractéristiques de type CI 41
4.1.2.3 Vecteur de caractéristiques de type patron 41
4.2 Étape 1 : entraînement 42
4.2.1 Entraînement du classificateur des CIUs 42
4.2.2 Entraînement du classificateur des CIs 43
4.2.3 Entraînement du classificateur de patrons 43
4.3 Étape 2 : détection 43
4.4 Métriques proposées 44
4.4.1 Métriques d'interaction avec l'utilisateur 44
4.4.1.1 Nombre d'évènements d'entrée basés sur des écouteurs (NLIE) 45
4.4.1.2 Nombre de gestionnaires d'évènements non liés au cycle de vie (NNLCEH) 45
4.4.2 Métriques de manipulation de la logique métier 46
4.4.2.1 Nombre de gestionnaires d'évènements du cycle de vie (NLCEH) 47
4.4.2.2 Nombre d'appels directs des CMs (NDCM) 47
4.4.2.3 Nombre d'appels indirects des CMs (NICM) 49
4.5 Données d'apprentissage 50
4.6 Validation et expérimentations 51
4.6.1 Questions de recherche 52
4.6.2 Algorithmes de classification 53
4.6.3 Données de validation 53
4.7 Résultats et discussions 54
4.7.1 QR1 : les métriques introduites sont-elles pertinentes pour la classification basée sur les rôles ? 54
4.7.2 QR2 : l'approche Coach basée sur la classification est-elle plus performante que l'approche Rimaz basée sur des heuristiques pour la détection automatique des patrons de type MVC ? 57
4.8 Conclusion 60

CHAPITRE 5 DÉTECTION DES PATRONS DE TYPE MVC DANS LES APPLICATIONS ANDROID À L'AIDE DES TECHNIQUES DE CLASSIFICATION MULTI-LABEL : L'APPROCHE MLCOACH 63
5.1 Étape 1 : entraînement 64
5.1.1 Entraînement du classificateur des classes modèles 65
5.1.1.1 Entrée 65
5.1.1.2 Sortie 66
5.1.1.3 Objectif 66

5.1.1.4 Méthodologie 66
5.1.2 Entraînement du classificateur des autres rôles 68
5.1.2.1 Entrée 68
5.1.2.2 Sortie 68
5.1.2.3 Objectif 68
5.1.2.4 Méthodologie 69
5.1.3 Entraînement du classificateur de patrons 71
5.1.3.1 Entrée 71
5.1.3.2 Sortie 71
5.1.3.3 Objectif 72
5.1.3.4 Méthodologie 72
5.2 Étape 2 : détection 73
5.2.1 Entrée 73
5.2.2 Sortie 73
5.2.3 Objectif 74
5.2.4 Méthodologie 75
5.3 Validation et expérimentations 75
5.3.1 Ensemble de données MlCoach 77
5.3.2 Données de validation 79
5.3.3 Métriques d'évaluation 79
5.4 Résultats et discussions 80
5.4.1 QR1 : Comment la classification multi-label se comporte-t-elle dans la détection de combinaison de patrons de type MVC ? 80
5.4.1.1 Performance de la classification des classes modèles 80
5.4.1.2 Performance de la classification des autres rôles 81
5.4.1.3 Performance de la classification de patrons 83
5.4.1.4 Comparaison de MlCoach, Coach et Rimaz 84
5.4.2 QR2 : Comment plusieurs patrons de type MVC peuvent-ils co-exister au sein d'une même application ? 86
5.4.2.1 QR2.1 : Quels patrons de type MVC ont tendance à coexister dans les applications Android ? 86
5.4.2.2 QR2.2 : Quels sont les facteurs les plus fréquents qui conduisent à l'apparition de multiples patrons de type MVC dans la même application ? 89
5.5 Conclusion 92

CHAPITRE 6 MENACES À LA VALIDITÉ 95
6.1 Validité de construction 95
6.2 Validité interne 95
6.3 Validité externe 96

CONCLUSION ET RECOMMANDATIONS 99

ANNEXE I CLASSIFICATION MULTI-LABEL 101

BIBLIOGRAPHIE 105

LISTE DES TABLEAUX

Page

Tableau 3.1 Synthèse des approches de détection automatique des patrons, d'anti-patrons et défauts de code dans les systèmes logiciels et mobiles 34

Tableau 4.1 Les métriques OO ajoutées 40

Tableau 4.2 Les métriques niveau application 41

Tableau 4.3 Distribution des patrons de type MVC dans l'ensemble de données d'apprentissage 50

Tableau 4.4 Nombre de classes significatives dans l'ensemble de données d'apprentissage 51

Tableau 4.5 Distribution des patrons de type MVC dans l'ensemble de données de validation 54

Tableau 4.6 Résultats de la classification des CIUs 55

Tableau 4.7 Résultats de la classification des CIs 56

Tableau 4.8 Résultats de la classification de patrons 57

Tableau 4.9 Les résultats obtenus par Coach et Rimaz en termes de précision, de rappel et de F1 58

Tableau 4.10 Les résultats obtenus par Coach et Rimaz en termes de nombre d'applications 58

Tableau 5.1 Métriques de code source introduites 71

Tableau 5.2 Métriques utilisées pour la classification de patrons 74

Tableau 5.3 Distribution des classes dans l'ensemble de données DCM 78

Tableau 5.4 Distribution des classes avec d'autres rôles dans l'ensemble de données DAR 78

Tableau 5.5 Distribution des patrons dans l'ensemble de données DP 78

Tableau 5.6 Performance globale de la classification des classes modèles 81

Tableau 5.7 Performance de la classification des autres rôles : évaluation par label 83

Tableau 5.8 Performance globale de la classification des autres rôles 83

Tableau 5.9 Performance de la classification de patrons : évaluation par label 84

Tableau 5.10 Performance globale de la classification de patrons 84

Tableau 5.11 Comparaison de MlCoach, Coach et Rimaz 85

Tableau 5.12 Les résultats de l'analyse pour QR2 88

LISTE DES FIGURES

Page

Figure 2.1 L'architecture de la plateforme Android, tiré de la Documentation-Android (2022) 9

Figure 2.2 Exemple d'un fichier de mise en page associé à une activité 11

Figure 2.3 Le cycle de vie d'une activité, tiré de la Documentation-Android (2021a) 12

Figure 2.4 Diagramme du patron architectural MVC 16

Figure 2.5 Diagramme du patron architectural MVP 17

Figure 2.6 Diagramme du patron architectural MVVM 18

Figure 2.7 Les deux implémentations les plus populaires du patron MVC dans la plateforme Android : Implémentation 1 19

Figure 2.8 Les deux implémentations les plus populaires du patron MVC dans la plateforme Android : Implémentation 2 19

Figure 2.9 Implémentation du patron MVP en Android 20

Figure 2.10 Implémentation du patron MVVM en Android 21

Figure 4.1 Aperçu de l'approche Coach 36

Figure 4.2 L'étape de prétraitement 36

Figure 4.3 Classification des métriques de code source 39

Figure 4.4 Exemple d'enregistrement d'évènement d'entrée, tiré de la Documentation-Android (2021b) 46

Figure 5.1 Aperçu de l'approche MlCoach 64

Figure 5.2 Aperçu de la tâche de classification des classes modèles 67

Figure 5.3 Aperçu de la tâche de classification des autres rôles 70

Figure 5.4 Aperçu de la tâche de classification de patrons 73

Figure 5.5 Processus de l'analyse qualitative 76

Figure 5.6 Distribution des patrons de type MVC parmi les applications étudiées selon les résultats de MlCoach 87

Figure 5.7 Les patrons appliqués dans les versions $V_{antrieure}$ et $V_{actuelle}$ des applications étudiées 89

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

ANN	Artificial Neural Network
API	Application Programming Interface
APK	Android PacKage
ART	Android RunTime
CNN	Convolutional Neural Network
GQM	Goal Question Metric
LOC	Lines Of Code
MLP	Multi-Layer Perceptron
MVC	Modèle Vue Contrôleur
MVP	Modèle Vue Présentateur
MVVM	Modèle Vue Vue-Modèle
OO	Orienté Objet
ORM	Object Relational Mapping
SDK	Software Development Kit
SOM	Self Organizing Map
SVM	Support Vector Machine
VIPER	Vue Interacteur Présentateur Entité Routeur

CHAPITRE 1

INTRODUCTION

1.1 Contexte et problématique

Le marché des applications mobiles est en plein essor, avec des milliards d'appareils, d'utilisateurs et d'applications. Survivre sur un tel marché nécessite une attention particulière à la qualité des applications, en particulier aux exigences non fonctionnelles telles que la maintenabilité, l'évolutivité et la testabilité (Verdecchia, Malavolta, Lago *et al.*, 2019). Ces dernières sont indispensables pour assurer une meilleure adaptation à l'évolution régulière des applications mobiles. De fait, plus de 29 % des applications Android les plus utilisées dans les magasins d'applications sont mises à jour chaque semaine pour répondre aux nouveaux besoins des utilisateurs et s'adapter à la mise à jour des environnements d'exécution (42matters, 2020a; Statista, 2020). Cela permet également de raccourcir les délais de mise sur le marché et réduire les coûts de maintenance et d'évolution. D'ailleurs, les développeurs Android accordent plus d'importance à la maintenabilité qu'à la performance quand ils conçoivent leurs applications (Verdecchia *et al.*, 2019) bien que l'aspect performance soit critique dans ces dernières. De plus, les applications mobiles sont des systèmes logiciels complexes (Bagheri, Garcia, Sadeghi, Malek & Medvidovic, 2016; Wasserman, 2010). À mesure que les capacités des appareils mobiles, telles que la puissance de calcul et la durée de vie de la batterie, continuent de s'améliorer, les besoins des utilisateurs évoluent également. Surtout avec l'émergence d'un grand nombre de nouvelles technologies telles que la réalité augmentée, la réalité virtuelle et l'intelligence artificielle, elles deviennent de plus en plus complexes.

Dans ce contexte, les développeurs sont amenés à concevoir correctement leurs applications en respectant les principes de la conception orientée objet et en appliquant les bonnes pratiques de développement, en particulier les patrons architecturaux. En effet, la nature interactive des applications mobiles a mené à l'adoption de certains patrons architecturaux adaptés aux systèmes interactifs, notamment le populaire Modèle-Vue-Contrôleur (MVC) et ses variantes Modèle-Vue-Présentateur (MVP) et Modèle-Vue-Vue-Modèle (MVVM), les patrons les plus utilisés

en développement Android (Verdecchia *et al.*, 2019; Daoudi, ElBoussaidi, Moha & Kpodjedo, 2019; Campos, Kulesza, Coelho, Bonifácio & Mariano, 2015).

Conçus pour répondre aux changements de l'interface utilisateur dans les systèmes interactifs, les patrons de type MVC proposent une décomposition particulière assurant le découplage des fonctionnalités noyau d'un système logiciel de son interface utilisateur. Elle consiste à structurer le système en trois principaux composants : un modèle qui représente la logique métier, une vue qui représente l'interface utilisateur et un autre composant qui agit comme intermédiaire entre les deux précédents. Une telle décomposition permet de construire des systèmes faciles à tester, à étendre, à entretenir et à réutiliser.

Les patrons de type MVC sont largement appliqués en développement Android (Verdecchia *et al.*, 2019; Daoudi *et al.*, 2019). Bien qu'ils soient très populaires, leur implémentation est toujours un sujet de discussion dans la communauté de développeurs. En fait, plusieurs implémentations ont été proposées dans les blogues et les sites de question-réponse tels que (Stackoverflow, 2010; Wang, 2020) qui discutent la meilleure façon de les mettre en place en fonction des composants de la plateforme. En pratique, la traduction de ces patrons en termes de composants spécifiques à cette plateforme n'est pas évidente. Certains composants, dans leurs définitions d'origine, sont ambigus et ne respectent pas les principes de la conception orientée objet (Bagheri *et al.*, 2016). Ensuite, les bonnes pratiques d'implémentation de ces patrons architecturaux ne sont pas toujours faciles à traduire en code. Certaines bonnes pratiques sont très génériques et abordent des principes de conception de base tels que la séparation de responsabilités et la décomposition (Verdecchia *et al.*, 2019). Même les différentes bibliothèques disponibles opérant au niveau architectural peuvent affecter leurs implémentations, car chaque bibliothèque spécifie ses composants et ses règles, et souvent, ils varient d'une bibliothèque à l'autre. En outre, les implémentations les plus couramment utilisées aujourd'hui peuvent évoluer ou devenir moins utilisées dans un proche avenir. En fait, les mises à jour régulières des plateformes de développement et des bibliothèques poussent les développeurs à suivre la communauté et à adopter les nouvelles implémentations. Dans ce cas, les développeurs se trouvent avec de multiples implémentations possibles. Le choix de la meilleure implémentation est ainsi une tâche

ardue, en particulier pour les développeurs novices. En pratique, les développeurs s'appuient sur leur expérience et leur compréhension des patrons architecturaux afin de choisir l'implémentation qui leur convient, et ce choix est souvent influencé par des considérations technologiques plutôt que par des considérations objectives. Dans cette optique, il devient nécessaire d'analyser les applications mobiles et de détecter automatiquement les différentes implémentations existantes des patrons de type MVC afin d'identifier les bonnes implémentations à suivre et les mauvaises à éviter en termes d'exigences de qualité, et mettre en évidence les violations récurrentes dans leur implémentation. Cela permettra également de comprendre les pratiques actuelles en matière de conception d'applications mobiles et de constituer une base de connaissances sur l'adoption de patrons architecturaux dans les applications mobiles.

Une autre problématique importante à laquelle nous nous intéressons dans ce travail est celle de la combinaison de multiples patrons de type MVC au sein de la même application mobile. La combinaison de plusieurs patrons de type MVC n'est pas un problème en soi, car une application peut avoir des préoccupations différentes qui nécessitent des solutions de conception différentes. Cependant, pour faire évoluer une application de manière cohérente, le développeur doit être conscient des patrons appliqués/combinés et des préoccupations qui ont conduit à ces patrons. Dans ce contexte, il devient important de fournir des outils qui peuvent aider les développeurs à identifier les patrons de type MVC appliqués dans leurs applications et à comprendre les problèmes de la conception actuelle et les facteurs qui les ont conduits.

À propos des travaux de recherche à cet égard, quelques travaux se sont intéressés à l'analyse d'architecture (Bagheri *et al.*, 2016; Campos *et al.*, 2015) et de bonnes pratiques de la conception architecturale des applications Android (Verdecchia *et al.*, 2019). Toutefois, la détection automatique des patrons architecturaux dans les applications Android n'est pas suffisamment abordée. Daoudi *et al.* (2019) ont proposé RIMAZ, une approche automatique basée sur des heuristiques pour la détection du patron de type MVC dominant dans les applications Android. Récemment, Komolov, Dlamini, Megha & Mazzara (2022) ont appliqué les techniques de classification pour la détection des patrons MVP et MVVM dans les applications Android. Bien que les deux approches proposées apportent des contributions intéressantes, elles se

limitent à un nombre réduit d'implémentations et à l'information structurelle extraite du code source. De plus, l'approche RIMAZ souffre du choix manuel des valeurs seuils utilisées par les heuristiques définies. Pour notre deuxième problématique, la situation est similaire. Malgré son importance, aucun travail n'a été effectué dans la littérature ciblant l'analyse et la détection des implémentations combinant multiples patrons architecturaux. Cela nous motive davantage à l'étudier dans les applications mobiles et à aider les développeurs mobiles à agir de manière appropriée.

1.2 Objectifs de la recherche

À long terme, le but principal de cette recherche est de contribuer à l'amélioration continue de la qualité des applications mobiles au niveau architectural en étudiant les patrons architecturaux dans les applications mobiles et en fournissant un outil de détection automatique pour soutenir les développeurs dans l'analyse de la qualité de leurs applications.

Cet objectif sera atteint en s'attaquant aux objectifs à court terme suivants :

- Proposer une approche de détection automatique des patrons de type MVC dans les applications Android indépendamment de leur implémentation ;
- Proposer une approche de détection automatique des implémentations combinant de multiples patrons de type MVC au sein de la même application dans les applications Android.

1.3 Contributions

Les contributions principales de notre travail sont les suivantes :

- Une liste de métriques de code source spécifiques à Android permettant de mesurer l'interaction avec l'utilisateur, la manipulation de la logique métier et la logique de présentation dans les applications Android ;
- L'ensemble de données MLCOACH contenant 69 applications Android annotées manuellement par le(s) patron(s) de type MVC appliqué(s), ainsi que le(s) rôle(s) de chaque classe ;

- L’approche Coach basée sur la classification pour la détection des patrons de type MVC dans les applications Android indépendamment de leur implémentation ;
- L’approche MlCoach basée sur la classification multi-label en utilisant l’apprentissage profond pour la détection de la combinaison de multiples patrons de type MVC dans les applications Android ;
- Et une analyse qualitative de l’historique de huit applications Android combinant multiples patrons de type MVC.

Au-delà de l’application des techniques d’apprentissage automatique, notre recherche est la première qui s’intéresse à l’étude de la combinaison de plusieurs patrons architecturaux dans les applications mobiles et enquête sur les facteurs qui y conduisent. Notre recherche est bénéfique aux développeurs Android dans la compréhension et l’évolution correcte de leurs applications d’un point de vue architectural, et aux chercheurs dans l’établissement de caractérisation objective des pratiques actuelles en matière de conception des applications mobiles.

1.4 Publications

Ces travaux de recherche ont fait l’objet d’une publication dans une conférence et d’une soumission à un journal, qui sont les suivantes :

- Chekhaba, Rebatchi, ElBoussaidi, Moha & Kpodjedo ***Coach : classification-based architectural patterns detection in Android apps.*** *36th ACM/SIGAPP Symposium On Applied Computing (ACM SAC 2021)*
- Chekhaba, El Boussaidi, Heminna, Moha & Desrosiers ***Detecting Multiple Architectural Patterns In Mobile Apps : A Multi-label Classification based Approach***, soumis au journal scientifique *Journal of Systems and Software (JSS).*

1.5 Paquet de réplication

Tous les outils, les données et les résultats de nos expérimentations sont disponibles sur :

- L’approche Coach https://github.com/coachAndroid/Coach
- L’approche MlCoach : https://github.com/ChaimaChekhaba/MLCOACH

1.6 Organisation du mémoire

La suite du document est organisée comme suit : dans le chapitre 2, nous introduisons les concepts de base sur lesquels s’appuie notre travail. Ensuite, dans le chapitre 3, nous présentons une revue critique de la littérature. Les chapitres 4 et 5 présenteront nos contributions : les approches Coach et MlCoach. Dans le chapitre 6, nous exposons les menaces potentielles qui pourraient influencer la validité de nos résultats. Et nous terminons par une conclusion et travaux futurs.

CHAPITRE 2

CONCEPTS DE BASE

Dans ce chapitre, nous présentons des concepts préliminaires sur lesquels s'appuie notre travail. D'abord, nous introduisons les concepts de base de la plateforme Android ainsi que les composants constituant toute application Android. Ensuite, nous décrivons les patrons architecturaux, les patrons de type MVC ainsi que leurs implémentations en Android.

2.1 Plateforme Android

Android est un système d'exploitation libre basé sur le noyau Linux, conçu pour une variété de dispositifs mobiles tels que les téléphones intelligents, les tablettes et les téléviseurs intelligents. Depuis son lancement en 2008, il continue de dominer le marché mondial des systèmes d'exploitation mobiles avec plus de deux milliards d'utilisateurs (Statista, 2022b). Son grand succès incite les développeurs à le choisir comme la première plateforme de développement d'applications mobiles. Dans ce travail, nous avons décidé de mener nos recherches sur cette plateforme, car elle est majoritairement à code source ouvert. De plus, l'accès à ses applications est plus simple par rapport à ses concurrents. Cela en fait la meilleure plateforme pour étudier les pratiques de développement mobile.

Les applications Android sont généralement des systèmes logiciels légers adaptés aux dispositifs mobiles sur lesquelles elles s'exécutent. Elles doivent être développées compte tenu des contraintes matérielles des dispositifs mobiles telles que la consommation énergétique, la puissance de calcul et la quantité de mémoire. Ces applications sont majoritairement écrites en Java ou Kotlin et distribuées via des magasins d'applications tels que Google Play. Le développement de ces applications s'appuie sur le cadre d'application Android. Chaque développeur doit être familiarisé avec les composants de base et ce que le cadre d'applications Android fournit afin de créer des applications robustes et de haute qualité.

Dans les sections suivantes, nous présentons quelques concepts clés au développement de toute application Android, à savoir l'architecture de la plateforme et les principaux composants d'une application Android.

2.1.1 Architecture et principaux composants

Illustrée dans la figure 2.1, l'architecture d'Android se compose de différentes couches permettant de répondre aux besoins de tout dispositif Android. Les principales couches sont les suivantes :

1. Le *noyau Linux* est le cœur de la plateforme Android. Il fournit une couche d'abstraction entre le matériel du dispositif et les autres couches de l'architecture ;
2. La *couche d'abstraction matérielle (Hardware Abstraction Layer)* fournit des interfaces standards exposant les capacités matérielles du dispositif aux couches supérieures ;
3. L'*environnement d'exécution d'Android* contient la machine virtuelle ART [1] (ou son prédécesseur Dalvik) qui se charge d'exécuter le code intermédiaire (*Bytecode*) généré à la compilation ;
4. Les *bibliothèques de la plateforme* comprennent diverses bibliothèques de base C/C++ et de Java telles que *Media* et *Surface Manager*, qui fournissent un support au développement Android ;
5. Le *cadre d'API Java* (ou le *cadre d'applications Android*) est le cœur du développement Android. Il fournit les composants de base sur lesquels repose toute application Android. Il comprend également un ensemble de services de gestion de ses composants et des ressources/fonctionnalités du système tels que le *gestionnaire d'activités* et les *fournisseurs de contenu* ;
6. La *couche d'applications* contient les applications natives et tierces installées sur le dispositif.

Dans ce travail, deux couches nous intéressent en particulier : la couche du cadre d'applications Android (5) nécessaire à la création de toute application Android, et la couche d'applications (6), contenant les applications conçues par les développeurs Android.

[1] Android RunTime (ART) and Dalvik : https ://source.android.com/devices/tech/dalvik

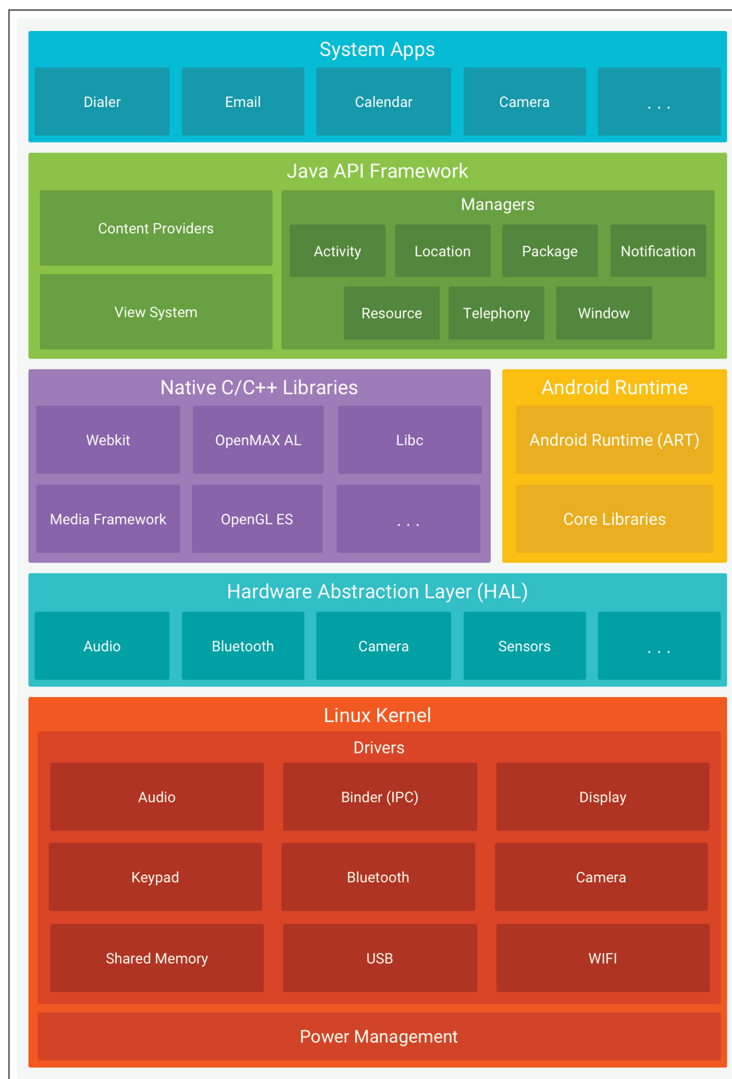

Figure 2.1 L'architecture de la plateforme Android, tiré de la Documentation-Android (2022)

Le cadre d'applications Android fournit les éléments constitutifs essentiels d'une application Android, appelés composants d'applications (*App components*). Ces composants servent de points d'entrée par lesquels le système ou l'utilisateur peut accéder à l'application. Chaque composant a des tâches spécifiques à accomplir et possède son propre cycle de vie. Il existe quatre principaux types de composants d'application :

- *Activité* ou (*Activity*) est le composant clé dans une application Android. Elle représente un écran unique d'une application avec laquelle un utilisateur peut interagir,

- *Service* permet à l'application de continuer à s'exécuter en arrière-plan pour effectuer des opérations de longue durée ou pour effectuer des travaux pour des processus distants ;
- *Fournisseur de contenu* ou (*Content Provider*) est responsable de la gestion de l'accès aux données partagées ;
- *Récepteur de diffusion* ou (*Broadcast Receiver*) se charge de répondre aux messages de diffusion provenant d'autres applications ou du système.

Les composants d'application sont regroupés dans un fichier XML appelé *AndroidManifest.xml*. Ce dernier décrit chaque composant et la manière dont ils interagissent entre eux (Documentation-Android, 2022). Il contient également d'autres informations sur l'application telles que les métadonnées, les exigences de la plateforme, les bibliothèques externes et les permissions requises.

En plus des *Activités*, le cadre d'application Android fournit d'autres composants dotés d'une interface graphique qui permettent l'interaction avec l'utilisateur. Ces composants sont de niveau plus granulaire et nécessitent d'être attachés aux *Activités*. En particulier, les *Fragments*, qui peuvent être considérés comme de petites activités utilisées pour décomposer les *Activités* en composants encapsulés et réutilisables. On trouve également les *Dialogues*, qui sont de petites formes avec leurs propres interfaces graphiques séparées de celle des *Activités* et s'affichent sous forme des fenêtres contextuelles. À un niveau encore plus granulaire se trouvent les *Vues* (ou *Views*) telles que les boutons et les champs de texte, qui représentent les éléments de construction de base de toute interface utilisateur.

Les composants *Activités*, *Fragments*, *Dialogues* et les *Vues* personnalisées peuvent déclarer leur contenu et leur disposition dans des fichiers de mise page ou (*Layout*). Ce sont des fichiers XML qui spécifient la structure de l'interface graphique, ses composants, ainsi que leur style (taille, marges, remplissage, etc.) (Documentation-Android, 2022). Les fichiers de mise en page sont attachés aux composants lors de l'initialisation à l'aide de la méthode `setContentView()`. Figure 2.2 illustre un exemple d'un fichier de mise en page associé à une activité.

```xml
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="112dp"
        android:text="@string/textview"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <Button...>

    <ImageView...>

</androidx.constraintlayout.widget.ConstraintLayout>
```

Figure 2.2 Exemple d’un fichier de mise en page associé à une activité

Certains composants du cadre d’applications Android sont impliqués dans la gestion du cycle de vie des applications, notamment les composants *Activité*, *Service* et *Récepteur de diffusion* (Documentation-Android, 2022). En fait, chaque composant fournit des méthodes de retour ou (*Callbacks*) pour gérer son cycle de vie en capturant les différents états de la création à la destruction. Ces méthodes doivent être bien comprises par les développeurs afin d’éviter que l’application soit tuée par le système. À titre d’exemple, les méthodes de retour gérant le cycle de vie des *Activités* sont les suivantes, comme illustrée dans la figure 2.3 :

- `onCreate()` est déclenchée lorsque le système crée l’activité pour la première fois. Cette méthode doit implémenter la logique de démarrage de l’application de base qui ne doit se produire qu’une seule fois au cours du cycle de vie de l’activité, telle que la création des vues et la liaison de données à des listes.
- `onStart()` est invoquée lorsque l’activité est rendue visible à l’utilisateur.
- `onResume()` est appelée lorsque l’activité passe au premier plan et devient interactive. Dans cet état, l’activité est au sommet de la pile des activités et y reste jusqu’à ce qu’un évènement

susceptible de l'interrompre soit déclenché, tel que la navigation de l'utilisateur vers une autre activité et la réception d'un appel téléphonique.

- `onPause()` est invoquée lorsque l'activité passe en arrière-plan suite au lancement d'une nouvelle activité. À cet état, l'activité n'est pas détruite par le système, elle n'est pas visible pour l'utilisateur.
- `onStop()` est déclenchée lorsque l'activité a fini de fonctionner et qu'elle est sur le point d'être arrêtée. L'activité n'est plus visible pour l'utilisateur.
- `onDestroy()` est appelée avant que le système ne détruise l'activité. Cela peut se produire parce que l'activité est en train de se terminer ou parce que le système détruit temporairement cette instance de l'activité pour gagner de la place.

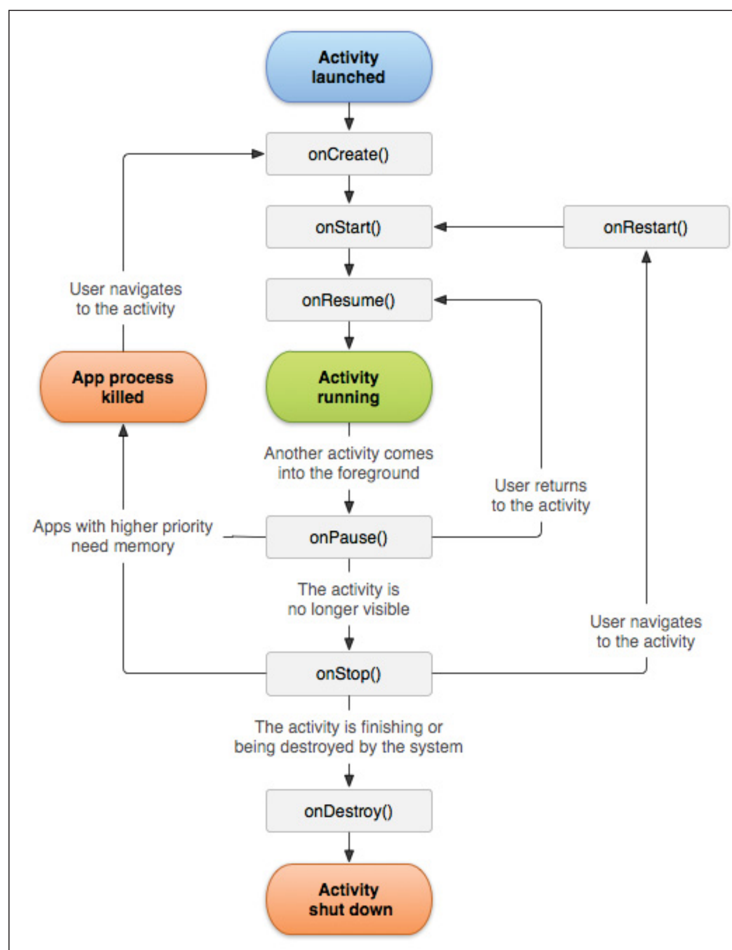

Figure 2.3 Le cycle de vie d'une activité, tiré de la Documentation-Android (2021a)

2.2 Patrons architecturaux

La notion de patrons architecturaux ou (*Architectural patterns*) est fondamentale en architecture logicielle. Ils fournissent des solutions génériques éprouvées à des problèmes spécifiques et récurrents en architecture logicielle. Selon Buschmann *et al.* (2008), “un patron architectural exprime un schéma d’organisation structurelle fondamental pour les systèmes logiciels en fournissant un ensemble de sous-systèmes prédéfinis, spécifiant leurs responsabilités et décrivant des règles et des directives pour organiser les relations entre eux”. Autrement dit, ce sont des patrons de haut niveau qui aident à spécifier la structure de base d’un système logiciel. Toutes les activités de développement ultérieures sont guidées par cette structure. Ils représentent des gabarits d’architectures logicielles concrètes (Buschmann *et al.*, 2008).

Chaque patron architectural aborde des exigences non fonctionnelles spécifiques à un système logiciel. Le patron MVC, par exemple, est conçu pour mieux gérer la variabilité dans les systèmes interactifs où le changement et l’adaptation de l’interface utilisateur sont très fréquents. Cela dit, leur adoption est très avantageuse pour répondre clairement et efficacement à ces besoins. Les patrons architecturaux aident également à bien documenter les architectures logicielles. L’utilisation d’un vocabulaire commun simplifie la compréhension des décisions architecturales et facilite la communication entre les parties prenantes, tout en leur permettant de gagner du temps dans la maintenance et l’évolution des systèmes. La gestion de la complexité des systèmes logiciels est un autre atout des patrons architecturaux. Suivre la décomposition offerte par un patron architectural dans la conception d’un système aide à réduire sa complexité et de la partager entre ses différents sous-systèmes. Ainsi, l’application des patrons architecturaux est fortement recommandée dans le développement logiciel.

En pratique, la plupart des systèmes logiciels ne peuvent pas être structurés suivant un seul patron architectural. Ils doivent répondre à plusieurs exigences non fonctionnelles souvent contradictoires qui ne peuvent être satisfaites que par des patrons architecturaux différents. Il est donc important de bien choisir le patron architectural ou la combinaison de patrons la plus

appropriée pour un système logiciel qui assure le meilleur compromis entre ses exigences. Il s'agit en effet d'une décision architecturale fondamentale.

Un autre concept proche aux patrons architecturaux est celui de patrons de conception ou *Design patterns*. Ces derniers sont définis comme des solutions génériques réutilisables à des problèmes de conception récurrents (Gamma, 1995). Ils décrivent généralement les relations et les interactions entre les classes et les objets et ont tendance à être indépendants des langages de programmation. Les patrons de conception permettent de construire une structure interne flexible, réutilisable et facile à maintenir. Comparés aux patrons architecturaux, ils sont plus petits, de portée moyenne et opèrent à un niveau d'abstraction moins élevé. À la différence des patrons architecturaux, leur application n'a aucun impact sur la structure de base d'un système logiciel, toutefois, elle peut influencer fortement l'architecture d'un sous-système (Buschmann *et al.*, 2008).

La plupart des patrons architecturaux soulèvent des problèmes qui peuvent être résolus par des patrons de conception. Le patron MVC, par exemple, utilise le patron de conception *Observateur* afin d'assurer la cohérence de l'état du modèle et les données à afficher dans les vues. D'autres patrons de conception sont également utilisés, tels que le patron *Composite* pour faciliter la gestion des composants de l'interface graphique et le patron *Stratégie* pour permettre l'échange des contrôleurs entre les vues.

Les patrons architecturaux sont largement appliqués en développement mobile. L'interactivité des applications mobiles a mené les plateformes de développement ainsi que les développeurs à adopter les patrons architecturaux adaptés aux systèmes interactifs, notamment le populaire MVC et ses variantes MVP et MVVM, et VIPER [2] (Verdecchia *et al.*, 2019; Daoudi *et al.*, 2019). D'autres patrons architecturaux ont été également utilisés par les développeurs mobiles, tels que *Architecture en couches* [3] et *Architecture hexagonale* [4] (Verdecchia *et al.*, 2019). Dans ce travail,

[2] VIPER : Pixelmatters (2020)

[3] Architecture en couches : Robert (2012)

[4] Architecture hexagonale : Martinez (2021)

nous nous concentrons sur les patrons les plus populaires en développement mobile, à savoir le patron MVC, MVP et MVVM. Les sections suivantes présentent chacun de ces patrons en détail.

2.2.1 Modèle Vue Contrôleur

Depuis son introduction dans les années 1970 par Krasner (1988), le patron MVC a connu un très grand succès dans le développement des systèmes interactifs. Il consiste à structurer le système en trois parties permettant une meilleure flexibilité dans l'adaptation de l'interface utilisateur. Les trois parties sont les suivantes :

- Le *Modèle* représente la couche métier du système. Il regroupe toutes les entités et les données qui représentent la connaissance, l'information, ainsi que la logique manipulant ces entités (Reenskaug, 1979). Le modèle doit être complètement indépendant des autres composants. Cependant, il peut notifier ses observateurs lorsqu'un changement de son état se produit.
- La *Vue* représente l'aspect visuel du système comme les boutons et les menus. Elle se charge de l'affichage des données provenant du modèle (Reenskaug, 1979; Dey, 2011). Plusieurs vues peuvent exister pour le même modèle et chaque vue est associée à un contrôleur.
- Le *Contrôleur* est le point d'entrée du système. Il gère l'interaction de l'utilisateur avec la vue telle que les cliques des boutons et la saisie de clavier en la traduisant en requêtes envoyées au modèle ou à la vue (Schmidt, Stal, Rohnert & Buschmann, 2013; Reenskaug, 1979). Un seul contrôleur peut être associé à plusieurs vues.

Figure 2.4 illustre les trois composants du patron MVC et les interactions entre eux. Au départ, la vue capture les évènements utilisateurs et les passe au contrôleur associé. Ce dernier se charge de les transformer en requêtes manipulant les entités du modèle ou des opérations d'affichage effectuées par la vue. Quand il y a un changement dans ses données, le modèle se charge de notifier les vues et les contrôleurs de son nouvel état. À leur tour, les vues récupèrent les nouvelles données du modèle et mettent à jour leur affichage.

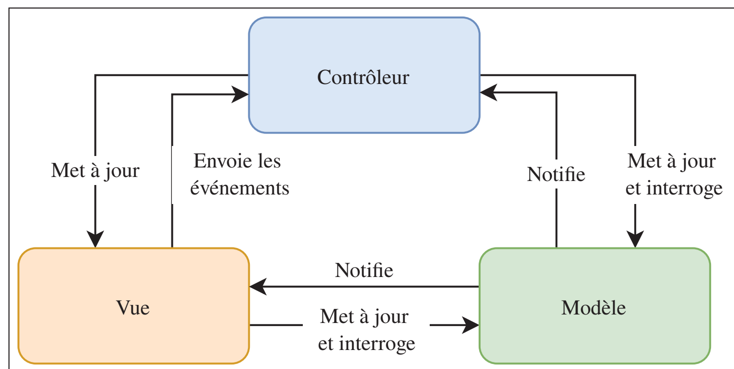

Figure 2.4 Diagramme du patron architectural MVC

Au fil des ans, le patron MVC a connu un grand succès et a été adapté à différents environnements. Sa mise en œuvre a évolué en conséquence, donnant naissance à d'autres variantes.

2.2.2 Modèle Vue Présentateur

Introduit dans les années 1990s, le MVP est une adaptation du patron MVC visant à apporter plus de découplage entre la couche présentation et la couche intermédiaire, représentée par les *présentateurs* (Sokolova, Lemercier, Garcia & Saint Luc, 2014).

Tout comme le patron MVC, le MVP décompose le système en trois composants : le *modèle*, la *vue* et le *présentateur*. Le modèle est défini de la même manière que dans le patron MVC. La *vue*, quant à elle, se charge de gérer les actions manipulant les évènements de l'utilisateur et les composants du système, et de visualiser les données du modèle. À la différence du contrôleur, le *présentateur* est responsable de la gestion des actions manipulant le modèle uniquement, en les transformant en requêtes manipulant les entités du modèle ou des opérations de mise à jour effectuées par la vue. En outre, le *présentateur* et la *vue* sont complètement découplés l'un de l'autre et communiquent via des interfaces.

Figure 2.5 illustre les trois composants du patron MVP et décrit les interactions entre eux. Tout d'abord, la vue reçoit les évènements de l'utilisateur et les transmet au présentateur au besoin, en vue de manipuler le modèle. À son tour, le présentateur reçoit les entrées de l'utilisateur, les

traite à l'aide des entités du modèle et renvoie les résultats à la vue. Cette dernière récupère les nouvelles données et met à jour son affichage.

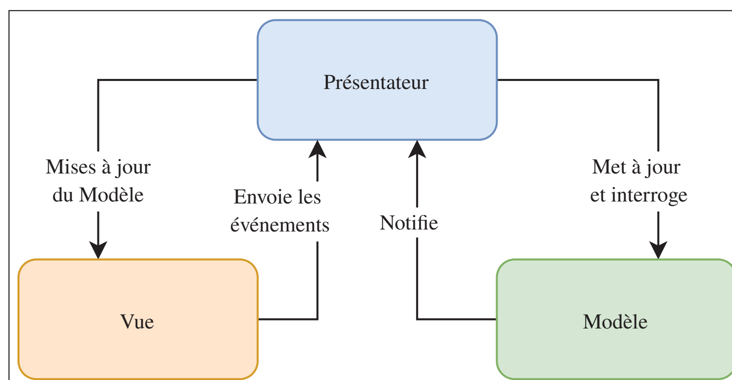

Figure 2.5 Diagramme du patron architectural MVP

Le patron MVP se présente sous deux variantes : *Contrôleur superviseur* et *Vue passive* (Sokolova *et al.*, 2014). Dans la variante *contrôleur superviseur*, la vue peut interagir avec le modèle directement afin d'enregistrer les données, uniquement lorsque aucune modification à apporter à la vue n'est requise. Dans ce cas, l'interaction se fait via le patron *Observateur*. Dans le cas contraire, la vue passe par le présentateur pour communiquer avec le modèle. Quant à la variante *vue passive*, l'interaction entre la vue et le modèle se fait exclusivement via le présentateur (Potel, 1996). C'est la variante la plus populaire en développement mobile.

2.2.3 Modèle Vue Vue-Modèle

Le patron MVVM, introduit par Microsoft en 2005, est une autre variante du MVC qui vise à découpler davantage la couche présentation de la couche métier (MSDN, 2019). Comme le montre la figure 2.6, le MVVM structure le système en trois composants : le *modèle*, la *vue* et le *vue-modèle*. Le modèle reste défini de la même manière que dans les deux autres patrons. La vue, de son côté, se limite à la visualisation des données du modèle. Le *vue-modèle*, comme son nom l'indique, n'est qu'un modèle de la vue. Il définit et implémente les évènements de l'utilisateur capturés par la vue, transforme les données du modèle et les rend prêtes à être affichées par la vue. Le *vue-modèle* est complètement indépendant de la vue. La vue, cependant, fait référence

au *vue-modèle*. La communication entre ces deux derniers se fait via le mécanisme de liaison de données bidirectionnelles (ou *Two-way Databinding*). Cela permet la propagation automatique des changements de l'état du *vue-modèle* à la vue.

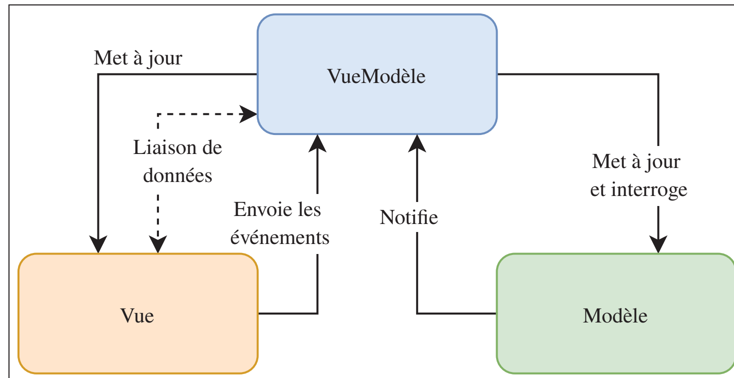

Figure 2.6 Diagramme du patron architectural MVVM

2.2.4 Implémentation des patrons de type MVC en Android

Dans les versions antérieures du SDK d'Android (avant 2017), aucune architecture n'était désignée comme architecture par défaut pour les applications Android. À cette époque, le patron MVC était le plus adopté par les développeurs Android (Bagheri *et al.*, 2016; Campos *et al.*, 2015). Cependant, son implémentation a toujours été un sujet de discussion dans la communauté de développeurs. Au cœur de cette discussion se trouve la traduction des composants de la plateforme possédant une interface graphique vers les composants du patron MVC. Certains développeurs considèrent les *activités* et les *fragments* comme appartenant à la vue, car ils représentent l'interface utilisateur de l'application. Dans ce cas, la partie contrôleur va être représentée par des classes indépendantes de la plateforme Android. D'autres développeurs pensent que les *activités* et les *fragments* sont à la fois des classes de la vue et du contrôleur. En effet, ce sont les classes dans lesquelles les évènements utilisateurs sont déclarés et gérés. Figures 2.7 et 2.8 illustrent les deux implémentations les plus populaires du patron MVC en termes de composants spécifiques à la plateforme Android.

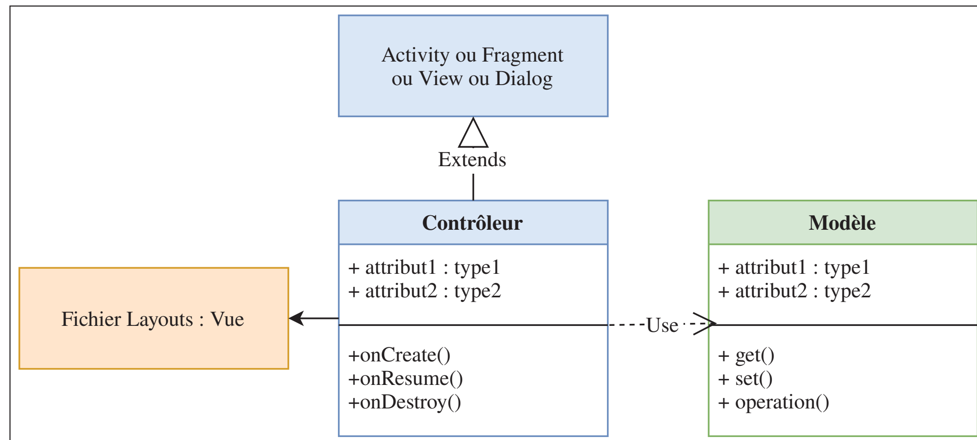

Figure 2.7 Les deux implémentations les plus populaires du patron MVC dans la plateforme Android : Implémentation 1

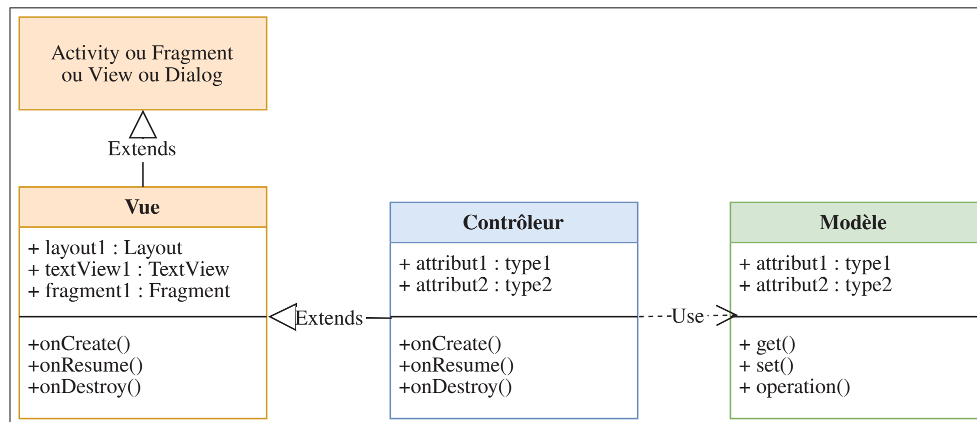

Figure 2.8 Les deux implémentations les plus populaires du patron MVC dans la plateforme Android : Implémentation 2

À la différence du MVC, le patron MVP a une traduction unique en composants de la plateforme Android. Plus précisément, les *activités* et les *fragments* correspondent à la partie vue, car ils représentent l'interface utilisateur, déclarent et gèrent les évènements utilisateurs. La partie présentateur est ainsi représentée par des classes indépendantes de la plateforme. Pour découpler le présentateur de la vue, la vue déclare un attribut de type présentateur abstrait. Ce dernier n'est qu'une classe abstraite qui déclare les méthodes de base que le présentateur concret doit redéfinir. De la même manière, le présentateur possède un attribut de type vue abstraite qui sera

redéfini par la vue concrète. Ce référencement croisé peut également être assuré par des classes génériques ou des interfaces. Figure 2.9 illustre l'implémentation de ce patron en termes de composants spécifiques à la plateforme Android.

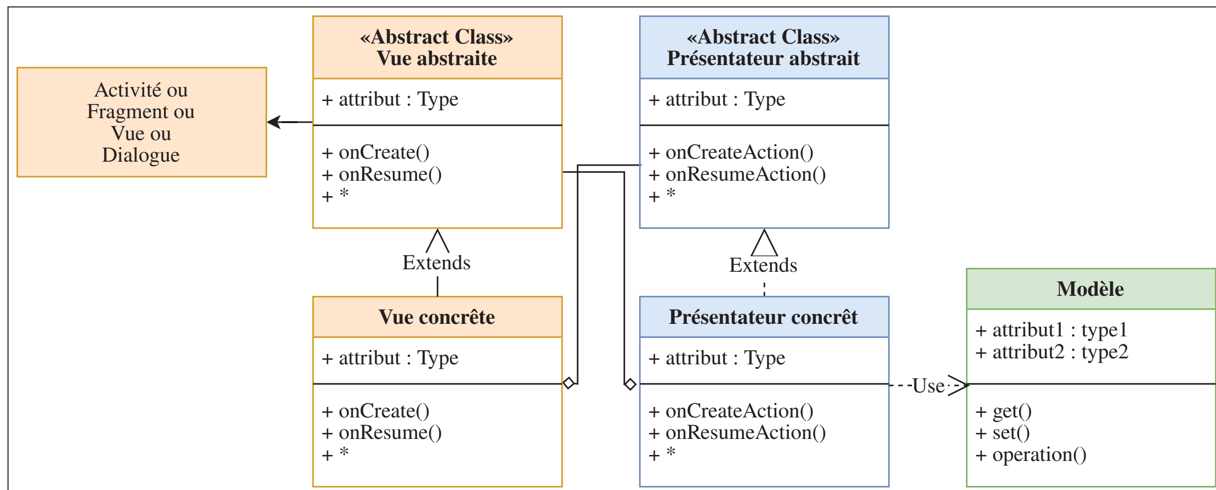

Figure 2.9 Implémentation du patron MVP en Android

Une telle implémentation présente une difficulté particulière liée à la gestion de cycle de vie des présentateurs. En effet, il n'est pas envisagé de répliquer le cycle de vie des composants d'Android aux présentateurs à partir de zéro. Cela demande plus de temps et d'efforts, et est souvent source des bogues. Toutefois, il est nécessaire d'assurer la cohérence entre le présentateur et la vue lorsque les composants de cette dernière sont détruits par le système. À cette fin, de nombreuses bibliothèques spécifiques à Android ont été proposées, notamment *Mosby (2018)* et *Moxy (2021)* pour simplifier sa mise en place.

Le patron MVVM, quant à lui, est désormais le patron recommandé pour les applications Android. Après l'introduction de la bibliothèque *Jetpack* en 2017, son implémentation a été simplifiée grâce à des composants tels que `AndroidViewModel` et `LiveData`. Tout comme le patron MVP, les *activités* et les *fragments* constituent la partie vue. Les classes `AndroidViewModel`, comme leur nom l'indique, représentent la partie vue-modèle qui assure la communication entre le modèle et la vue, tout en respectant le cycle de vie des classes de la vue. Les classes `LiveData` qui sont, à leur tour, conçues avec respect de cycle de vie des composants de la vue

et le vue-modèle, assurent la mise à jour bidirectionnelle automatique des données entre le modèle et le vue-modèle. Le mécanisme de liaison de données bidirectionnelle entre la vue et le vue-modèle est assuré automatiquement à travers la bibliothèque *Databinding*. Cette dernière permet d'associer les évènements utilisateurs aux méthodes qui les gèrent dans les fichiers de mise en page associés aux vues. Figure 2.10 illustre l'implémentation du patron MVVM en termes de composants spécifiques à la plateforme Android.

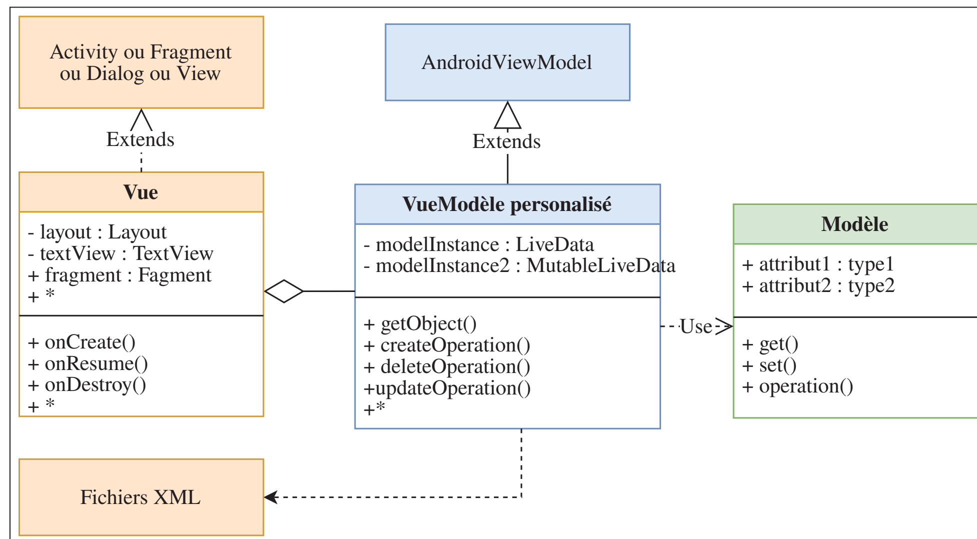

Figure 2.10 Implémentation du patron MVVM en Android

2.3 Conclusion

Dans ce chapitre, nous avons introduit les concepts de base nécessaires à la compréhension de notre travail, à savoir les principes de base de la plateforme Android, les patrons architecturaux et les patrons de type MVC. Nous avons également fait le point sur l'implémentation des patrons de type MVC dans la plateforme Android ainsi que le débat sur la meilleure implémentation en termes de composants de la plateforme. Le prochain chapitre expose les travaux effectués dans la littérature portant sur l'analyse et la détection automatique des patrons architecturaux en mettant l'accent sur les travaux appliquant l'apprentissage automatique. Nous présenterons aussi d'autres travaux intéressants sur des concepts similaires tels que la détection des patrons de conception, d'anti-patrons et défauts de code.

CHAPITRE 3

REVUE DE LITTÉRATURE

Dans ce chapitre, nous exposons les travaux liés à notre problématique de recherche. D'abord, nous présentons les travaux portant sur l'analyse d'architecture des applications mobiles en mettant l'accent sur les patrons architecturaux. Nous survolons ensuite les travaux liés à la détection automatique de patrons dans les systèmes logiciels et applications mobiles. Nous présentons également les travaux qui portent sur la détection automatique des concepts similaires, à savoir les *anti-patrons* et les *défauts de code* dans les systèmes logiciels et applications mobiles. Après, nous discutons les approches de détection automatique basées sur les techniques d'apprentissage automatique. Nous terminons avec une synthèse générale.

3.1 Analyse d'architectures et de patrons architecturaux dans les applications mobiles

Peu de travaux de recherches se sont intéressés à l'analyse d'architectures et de patrons architecturaux dans les applications mobiles. Bagheri *et al.* (2016) ont étudié l'adoption des principes de conception architecturale dans le SDK d'Android. Selon leurs résultats, les applications Android sont conçues en respectant plusieurs principes de conception, notamment la décomposition basée sur composants et connecteurs et l'application des styles architecturaux. De plus, une variété de styles architecturaux a été utilisée comme *l'invocation implicite basée sur les messages* entre les *activités* et les *services*, et *l'état partagé* entre les composants de type *fournisseur de contenu*. Les auteurs ont également exploré des centaines d'applications Android afin d'analyser les différentes architectures existantes. Ils ont conclu que les applications Android sont des systèmes logiciels complexes constitués de dizaines de composants et connecteurs et utilisent plusieurs styles architecturaux. Ils ont également identifié certaines violations courantes dans la conception des applications Android, en particulier les violations liées au cadre de développement. Selon leur analyse, les composants de base fournis sont strictement définis et capturent une sémantique réduite par rapport au monde réel. En outre, les composants *récepteurs de diffusion* sont à la fois définis comme des composants et des connecteurs (au sens

d'architecture logicielle). Les composants *activités*, quant à eux, ne respectent pas le principe de séparation de responsabilités vu qu'ils représentent l'interface graphique et déclarent et gèrent les évènements utilisateurs en même temps. Cela a conduit à une mauvaise implémentation du patron MVC, le plus populaire dans le développement mobile.

Une autre étude intéressante menée par Campos., Kulesza., Coelho., Bonifácio. & Mariano. (2015) analyse l'adoption des patrons architecturaux dans les applications Android. À travers une étude exploratoire, les auteurs ont analysé le code source de 12 applications Android populaires et libres, et ont examiné les dépendances entre les composants architecturaux de ces applications à l'aide de l'outil JDepend (Clark & Perez, 2022). Selon leurs résultats, MVC est le patron le plus utilisé dans le développement d'applications Android. De plus, la grande majorité des développeurs l'implémente de la même manière. Cette dernière consiste à mettre en place le modèle par des classes de type `Service`, `Data` ou `Entity`, la vue sous forme de fichiers XML ou des vues personnalisées qui étendent les vues de base fournies par la plateforme, et le contrôleur par des classes de type `Activity` et `Fragment`. Les auteurs ont également identifié deux violations relatives à la mise en place du composant modèle en MVC. D'après leur analyse, certaines classes modèles récupèrent des éléments de la vue dans le but de les mettre à jour. D'autres classes modèles référencent les classes contrôleurs et changent leurs états. Une telle implémentation viole l'indépendance du modèle des autres composants.

Un autre travail intéressant de Sokolova *et al.* (2014) analyse l'implémentation de certains patrons architecturaux largement utilisés dans les applications Android. Les auteurs ont discuté la difficulté de mettre en place les patrons de type MVC en Android, en particulier dans l'implémentation des *activités*. Selon leur analyse, les implémentations existantes traitent les *activités* comme faisant partie de la vue ou du contrôleur ou même du présentateur, et les chargent souvent de la logique de présentation et les actions liées aux modèles et à l'interaction avec l'utilisateur. En conséquence, elles deviennent rapidement volumineuses, très complexes et très difficiles à maintenir ou à étendre. Ils ont donc proposé le patron *Android Passive MVC*, une adaptation du patron MVC pour la plateforme Android. Ce dernier considère l'*activité* comme un composant intermédiaire entre la vue et le contrôleur. Chaque *activité* gère un ensemble de

contrôleurs en les associant aux vues correspondantes. Chaque contrôleur est responsable de la gestion des évènements utilisateurs et la logique de présentation d'un ensemble réduit de vues, facilitant ainsi leur réutilisation. La vue, quant à elle, est représentée, soit par les vues de base fournies par la plateforme, soit par des vues personnalisées qui les étendent. Cette extension a été appliquée à un exemple d'application de réseau social en collaboration avec un industriel, et s'est avérée efficace.

Récemment, Verdecchia *et al.* (2019) ont examiné les bonnes pratiques architecturales dans la conception d'applications Android. À travers une analyse empirique de la littérature, suivie d'une investigation auprès de cinq praticiens, les auteurs ont catalogué 42 bonnes pratiques spécifiques à Android liées à la conception d'applications Android en général et l'implémentation des patrons architecturaux, notamment le patron MVP, MVVM et l'architecture en couches. À titre d'exemple, l'une des bonnes pratiques liées à la mise en place du patron MVP est de mettre le présentateur dépendant des vues plutôt que des activités. Une autre bonne pratique consiste à mettre les présentateurs indépendants de la plateforme Android. Les auteurs ont également rapporté d'autres conclusions intéressantes liées à l'utilisation des bibliothèques facilitant la conception et l'implémentation des patrons architecturaux tels que *Mosby*, *RxJava* et *Jetpack* et les exigences de qualité qui préoccupent le plus les développeurs Android, à savoir la maintenabilité, la testabilité et la performance.

3.2 Détection automatique de patrons architecturaux dans les applications mobiles

Quelques travaux de recherches ont abordé le problème de détection automatique des patrons architecturaux dans les applications mobiles. Daoudi *et al.* (2019) ont proposé RIMAZ, une approche basée sur des heuristiques permettant d'identifier le patron de type MVC dominant dans les applications Android. Tout d'abord, RIMAZ récupère le code source de l'application à analyser depuis son APK en appliquant la rétro-ingénierie. Le code source généré est ensuite filtré en supprimant toutes les dépendances externes utilisées. Puis, à travers une analyse statique du code source, toutes les classes de l'application sont classées en quatre catégories selon leurs responsabilités. Les catégories considérées sont : les classes modèles (MCs), les classes

d’interactions (ICs), les classes de liaison de données (DBCs) et les classes auxiliaires (HCs). Les évènements qui manipulent les MCs sont ensuite identifiés en inspectant chaque appel de méthodes, d’attributs ou de classes à ces dernières à partir des ICs. Enfin, RIMAZ différencie entre les trois patrons selon l’emplacement des évènements manipulant les MCs, c.-à-d. la catégorie de classes qui les gèrent. Bien que l’approche RIMAZ propose une caractérisation pertinente des patrons de type MVC en termes d’évènements et manipulation des classes modèles tout en tenant compte les spécificités de la plateforme Android, elle se limite aux évènements basés sur des écouteurs déclarés explicitement dans les classes. Ils peuvent toutefois être déclarés dans les fichiers de mise en page des composants graphiques et associés aux méthodes qui les gèrent directement à travers la bibliothèque de liaison de données *Databinding (2016)*. Ce type d’évènements est souvent présent avec le patron MVVM. En outre, les heuristiques proposées nécessitent le choix manuel de certaines valeurs seuils utilisées dans la détection. Cela rend la détection biaisée et imprécise. Finalement, l’approche échoue dans la détection du patron MVVM. Or, ce dernier est actuellement la recommandation pour la conception des applications Android.

Un travail récent mené par Komolov *et al.* (2022) propose une approche basée sur la classification pour la détection des patrons MVP et MVVM dans les applications Android. Les auteurs ont exploité les métriques OO calculées au niveau des classes, des méthodes et des variables afin de caractériser les patrons étudiés. Ils ont expérimenté plusieurs algorithmes classiques d’apprentissage automatique. Les classificateurs de l’approche ont été entraînés sur un ensemble de données de 5 973 applications collectées à partir de GITHUB et labellisées avec le patron appliqué sur la base des tags fournis par les développeurs. Bien que l’étude présente des résultats prometteurs, elle ne permet pas la détection du MVC, le patron le plus dominant dans les applications Android (Daoudi *et al.*, 2019; Campos *et al.*, 2015). De plus, l’approche est limitée aux métriques OO calculées par une analyse statique.

3.3 Détection automatique d'anti-patrons et défauts de code dans les applications mobiles

Quelques travaux ont abordé le problème de détection automatique d'anti-patrons et de défauts de code dans les applications mobiles. Les anti-patrons sont des mauvaises solutions à des problèmes de conception récurrents (Koenig, 1995), tandis que les défauts de code sont des problèmes d'implémentation qui proviennent de mauvais choix conceptuels (Wells & Williams, 2003).

Carvalho, Aniche, Veríssimo, Durelli & Gerosa (2019) se sont intéressés à l'identification des défauts de code spécifiques à Android qui se manifestent à la couche présentation. Les auteurs ont d'abord proposé un catalogue de 20 défauts de code spécifiques à Android auxquels les développeurs donnent plus d'importance suite à une investigation auprès de 246 développeurs Android. Ensuite, ils ont proposé une approche basée sur des règles qui exploitent des propriétés calculées à partir du code source telles, que le nombre de fragments et le nombre de fichiers de mise en page. Ces règles sont ainsi utilisées pour identifier les défauts de code présents dans les applications Android.

Palomba, Di Nucci, Panichella, Zaidman & De Lucia (2017) ont proposé l'outil aDoctor permettant l'identification de 15 défauts de code spécifiques à Android à partir du catalogue de Reimann, Brylski & Aßmann (2014). L'outil aDoctor exploite l'arbre syntaxique abstrait (AST) du code source et l'explore en appliquant des règles de détection basées sur les définitions fournies dans le catalogue de Reimann *et al.* (2014). Les auteurs ont évalué leur outil sur un ensemble de données construit manuellement et ont obtenu d'excellents résultats comparés à l'état de l'art.

Kessentini & Ouni (2017) ont proposé une approche de génération automatique des règles de détection des défauts de code dans les applications Android à l'aide de la programmation génétique multi-objectifs. L'approche proposée cherche à trouver le meilleur ensemble de règles qui couvrent un ensemble d'exemples de défauts de code d'applications Android à partir de deux

fonctions objectives contradictoires de précision et de rappel. Les auteurs ont évalué l'approche sur un ensemble de 184 applications Android libres et ont obtenu de très bons résultats.

Pareillement, Hecht, Benomar, Rouvoy, Moha & Duchien (2015) ont proposé une approche outillée appelée Paprika visant l'identification de huit anti-patrons, dont quatre sont spécifiques à Android. À partir de l'APK d'une application Android, Paprika applique la rétro-ingénierie afin de récupérer le code source de l'application à l'aide de l'outil Soot (Soot, 2022). À travers une analyse statique, Paprika génère une représentation sous forme de graphe d'application et calcule un ensemble de métriques de code source qui constituent les caractéristiques des nœuds du graphe. Les métriques calculées sont ensuite exploitées afin d'identifier les anti-patrons et les défauts de code en utilisant un ensemble de règles de détection sous forme de requêtes sur les graphes d'applications.

3.4 Utilisation de l'apprentissage automatique pour la détection de patrons, d'anti-patrons et de défauts de code

Avec leur grand succès dans de nombreux domaines, les techniques d'apprentissage automatique ont été employées en génie logiciel pour la résolution de plusieurs problèmes, notamment la détection des patrons architecturaux, patrons de conception, d'anti-patrons et de défauts de code dans les systèmes logiciels. Cette section reportera quelques travaux connexes, en mettant l'accent sur la définition du problème, la caractérisation des patrons/anti-patrons/défauts de code à détecter, les algorithmes d'apprentissage automatique utilisés, ainsi que les données d'apprentissage.

3.4.1 Détection de patrons basée sur l'apprentissage automatique

Récemment, Guamán, Delgado & Pérez (2021) se sont intéressés à l'identification des caractéristiques de qualité qui établissent la similarité entre les applications Java appliquant le patron architectural MVC dans une perspective de qualité. À travers une étude exploratoire, les auteurs ont analysé 87 applications Java appliquant le patron MVC et classé leurs implémentations en 13 variantes. Ils ont ensuite appliqué la classification non supervisée sur l'ensemble des

implémentations en employant le modèle neuronal Self Organizing Map (SOM). À l'aide de l'outil SonarQube (SonarQube, 2022), une liste de 24 métriques de qualité calculées au niveau de l'application a été utilisée pour former des vecteurs de caractéristiques. Selon leurs résultats, les métriques mesurant la taille en termes de nombre de LOC et de déclarations (ou *Statement*), la maintenabilité et la complexité du code sont les principaux facteurs permettant de déterminer la similarité entre les applications Java adoptant le patron MVC. Les auteurs ont également conclu que la qualité de ces applications est indépendante des choix technologiques.

Pareillement, Dwivedi, Tirkey & Rath (2019) ont proposé une approche de détection de six patrons de conception basée sur la rétro-ingénierie et la classification. Chaque patron a été défini par un ensemble de rôles, où chaque rôle est représenté par 67 métriques OO. Les auteurs ont utilisé divers outils, tels que Marple (Zanoni, Fontana & Stella, 2015) et Ssa (Tsantalis, Chatzigeorgiou, Stephanides & Halkidis, 2006), pour construire les données d'apprentissage et identifier les instances des patrons de conception à partir de trois systèmes logiciels libres. Les instances identifiées ont été ensuite validées à travers une analyse manuelle par des experts. Pour la classification, les auteurs ont expérimenté plusieurs algorithmes tels que les réseaux de neurones artificiels (ANN) et la régression logistique. Ils ont obtenu des résultats intéressants.

Chihada, Jalili, Hasheminejad & Zangooei (2015) ont proposé une approche basée sur la classification permettant la détection de six patrons de conception à partir du code source. Les auteurs ont défini chaque patron par un ensemble de rôles attribués aux classes. Chaque rôle a été représenté par un vecteur de caractéristiques créé en utilisant une liste de 45 métriques OO. À l'aide de l'algorithme de classification SVM, ils ont construit un ensemble de classificateurs (un classificateur par patron) en les entraînant sur un ensemble d'implémentations de patrons de conception identifiées manuellement par des experts. Ils ont obtenu de bonnes performances et ont pu identifier différentes versions des patrons étudiés.

3.4.2 Détection d'anti-patrons et défauts de code basée sur l'apprentissage automatique

De même pour les anti-patrons et les défauts de code, quelques études ont été proposées qui exploitent les techniques de classification. Khomh, Vaucher, Guéhéneuc & Sahraoui (2011) ont présenté BDTEX (*Bayesian Detection Expert*), une approche basée sur la méthode Objectif-Question-Mesure (GQM) [5] et les réseaux bayésiens pour la détection de trois anti-patrons. Les auteurs ont utilisé la méthode GQM pour définir formellement les anti-patrons étudiés sous forme de symptômes en spécifiant les métriques OO impliquées. Ces définitions sont ensuite utilisées pour construire des réseaux bayésiens qui expriment la relation causale entre les anti-patrons, leurs symptômes et les seuils des métriques impliquées. Les auteurs ont expérimenté leur approche sur deux systèmes logiciels libres et ont obtenu de très bons résultats.

Un autre travail intéressant, réalisé par Maiga *et al.* (2012), propose l'approche SMURF, qui permet d'identifier quatre anti-patrons. Les auteurs ont exploité les métriques orientées objet afin de caractériser les anti-patrons étudiés au niveau des classes et ont appliqué l'algorithme SVM pour la classification. Ils ont entraîné les classificateurs sur trois systèmes logiciels et ont obtenu de très bonnes performances.

Encore, une autre étude menée par Amorim, Costa, Antunes, Fonseca & Ribeiro (2015), évalue l'efficacité des arbres de décision pour la détection des anti-patrons. Les auteurs justifient le choix cet algorithme par sa capacité à apprendre des règles de détection faciles à comprendre. De la même façon, ils ont exploité les métriques orientées objet pour caractériser les anti-patrons étudiés et ont obtenu de très bons résultats.

Fontana, Mäntylä, Zanoni & Marino (2016) ont mené une étude comparative à grande échelle sur l'efficacité des algorithmes d'apprentissage automatique pour la détection des anti-patrons. Ils ont expérimenté 16 algorithmes différents d'apprentissage automatique sur 74 systèmes logiciels appartenant à l'ensemble de données QUALITAS CORPUS (Tempero *et al.*, 2010). Ils

[5] Goal-Question-Metric (GQM) : est une méthode éprouvée pour mettre en place des mesures axées sur les objectifs dans une organisation logicielle. https://www.leadingagile.com/2017/05/agile-metrics-gqm-approach/

ont conclu que l'application de l'apprentissage automatique à la détection d'anti-patrons peut fournir une grande précision, même avec un nombre limité de jeux de données.

Plus récemment, quelques travaux de recherche ont appliqué les techniques d'apprentissage profond. Liu, Xu & Zou (2018) ont proposé une approche basée sur les réseaux de neurones convolutifs (CNNs) et le *plongement de mots* [6] visant la détection d'anti-patrons. Dans ce travail, les auteurs ont exploité deux types d'informations extraites du code source : les informations textuelles telles que les noms de méthodes et de classes, et structurelles comme les métriques orientées objet. Les informations textuelles sont ensuite transformées en vecteurs de caractéristiques à l'aide du modèle *Word2Vec* (Mikolov, Chen, Corrado & Dean, 2013). Après, les vecteurs obtenus sont injectés à CNN afin d'extraire automatiquement les caractéristiques les plus pertinentes à la détection. Les métriques orientées objet, quant à eux, elles sont alimentées à un autre CNN afin d'identifier automatiquement les métriques les plus pertinentes. À la fin, les résultats des deux CNNs sont combinés et injectés à un ANN qui se charge de la classification. Pour entraîner un tel classificateur, les auteurs ont proposé une méthode de génération automatique de données d'entraînement. Ils ont obtenu des résultats prometteurs et ont pu surpasser toutes les approches existantes.

Une autre étude récente proposée par Barbez, Khomh & Guéhéneuc (2019) utilise les CNNs pour la détection de l'anti-patron *composant Dieu*. Les auteurs ont également exploité deux types d'informations : structurelles à partir des métriques OO et historiques à travers l'exploitation de l'historique complète des systèmes étudiés. Chaque système est représenté par un ensemble de matrices dont chacune contient les métriques OO calculées pour une version du système. L'ensemble de matrices est ensuite alimenté à un CNNs qui se charge d'extraire les informations les plus pertinentes à la détection, c.-à-d. les métriques et la façon dont elles évoluent dans le temps. Pour entraîner leur classificateur, ils ont utilisé un ensemble de données analysé manuellement contenant des occurrences de défauts de conception étudiés dans huit projets Java libres. Ils ont également obtenu de très bons résultats, surpassant toutes les approches existantes.

[6] Word embedding : https://en.wikipedia.org/wiki/Word_embedding

3.5 Synthèse générale

Tous les travaux présentés apportent des contributions intéressantes au domaine du développement mobile. En particulier, les travaux de Bagheri *et al.* (2016); Campos. *et al.* (2015); Verdecchia *et al.* (2019) qui donnent un aperçu sur les pratiques actuelles en matière de conception architecturale des applications Android et l'emploi des patrons architecturaux. Ils fournissent également un aperçu sur les implémentations les plus populaires des patrons architecturaux en Android, les bonnes pratiques ainsi que les violations les plus fréquentes parmi les développeurs.

Concernant la détection automatique des patrons, d'anti-patrons et défauts de code dans les systèmes logiciels et mobiles, nous synthétisons les approches présentées ci-dessus dans le tableau 3.1. La majorité des études ciblant les applications Android proposent des approches basées sur des règles (Hecht *et al.*, 2015; Carvalho *et al.*, 2019) et des heuristiques (Daoudi *et al.*, 2019) et se limitent à l'information structurelle extraite à partir du code source à travers une analyse statique. Ce genre d'approches définissent des valeurs seuils identifiées empiriquement dans le processus de détection et par conséquent, leurs résultats sont étroitement liés aux valeurs choisies. D'où la nécessité d'appliquer les techniques d'apprentissage automatique.

À cet égard, la plupart des approches basées sur l'apprentissage automatique appliquent les techniques de classification supervisée avec différents algorithmes, tels que les SVM, les arbres de décisions et les réseaux de neurones artificiels (ANNs) en exploitant l'information structurelle (généralement les métriques OO). Bien que la plupart de ces approches soient entraînées sur des ensembles de données de petite taille, elles ont obtenu de meilleures performances et ont pu détecter les différentes implémentations/formes de patrons/anti-patrons/défauts de code dans les systèmes logiciels. De même, pour les travaux appliquant l'apprentissage profond, la majorité des approches exploitent plusieurs sources d'informations à partir du code source c.-à-d. l'information textuelle, structurelle et historique, et permettent d'obtenir de très bonnes performances comparées aux autres approches. Toutefois, l'apprentissage profond nécessite une quantité énorme de données, ce qui est très difficile à avoir. En outre, tous ces travaux ont été appliqués sur des ensembles de données différents, car il n'existe pas des données de référence

standards dans la littérature. Ainsi, répliquer ces techniques dans le contexte de mobile nécessite la construction manuelle des données d'apprentissage.

Tableau 3.1 Synthèse des approches de détection automatique des patrons, d'anti-patrons et défauts de code dans les systèmes logiciels et mobiles

Étude	Plateforme	Patrons/défauts visés	Approche	Type de caractéristiques
Komolov *et al.* (2022)	Android	Patrons architecturaux MVP et MVVM	Classification avec plusieurs algorithmes	Information structurelle
RIMAZ (Daoudi *et al.*, 2019)	Android	Patrons architecturaux de type MVC	Heuristiques	Information structurelle
Carvalho *et al.* (2019)	Android	Défauts de code dans la couche de présentation	Règles	Information structurelle
aDOCTOR (Palomba *et al.*, 2017)	Android	Défauts de code spécifiques à Android	Règles	Métriques de code source
Kessentini & Ouni (2017)	Android	Défauts de code spécifiques à Android	Programmation génétique multi-objectifs	Métriques de code source
PAPRIKA (Hecht, Moha & Rouvoy, 2016)	Android	Défauts de code spécifiques à Android + OO	Règles	Métriques de code source
Guamán *et al.* (2021)	Systèmes OO	Patron architectural MVC	Classification non supervisée avec le SOM	Métriques de code source
Dwivedi *et al.* (2019)	Systèmes OO	Patrons de conception	Classification avec les ANN et la régression logistique	Métriques de code source
Chihada *et al.* (2015)	Systèmes OO	Patrons de conception	Classification avec le SVM	Métriques de code source
Amorim *et al.* (2015)	Systèmes OO	Défauts de code OO	Classification avec les arbres de décision	Métriques de code source
BDTEX (Khomh *et al.*, 2011)	Systèmes OO	Défauts de code OO	Classification avec les réseaux bayésiens BBN	Métriques de code source
SMURF (Maiga *et al.*, 2012)	Systèmes OO	Anti-patrons	Classification avec le SVM	Métriques de code source
Fontana *et al.* (2016)	Systèmes OO	Anti-patrons	Classification avec plusieurs algorithmes	Métriques de code source
Liu *et al.* (2018)	Systèmes OO	Anti-patrons	Classification avec l'apprentissage profond (CNN)	Information textuelle + métriques de code source
Barbez *et al.* (2019)	Systèmes OO	Anti-patrons	Classification avec l'apprentissage profond (CNN)	Information historique + métriques de code source

CHAPITRE 4

DÉTECTION DES PATRONS DE TYPE MVC DANS LES APPLICATIONS ANDROID BASÉE SUR LA CLASSIFICATION : L'APPROCHE COACH

Dans ce chapitre, nous présentons notre première contribution principale : l'approche Coach (*Classification Oriented Approach for ArCHitectural pattern detection*). Cette dernière permet de détecter les patrons de type MVC dans les applications Android à travers les techniques de classification. Ces techniques ont été choisies en raison de leur grand succès dans plusieurs domaines, notamment dans l'identification des patrons de conception (Chihada *et al.*, 2015; Dwivedi *et al.*, 2019; Fontana *et al.*, 2016). Elles ont amélioré significativement les résultats de la détection en identifiant différentes implémentations de patrons de conception, le tout sans fournir des définitions statiques ou des formalisations.

Illustrée dans la figure 4.1, l'approche Coach se décline en deux étapes principales : l'entraînement et la détection, avec une étape préliminaire de prétraitement. L'étape de prétraitement exploite des informations spécifiques à la plateforme Android afin d'identifier les classes significatives des patrons de type MVC dans une application donnée et calcule une liste de métriques de code source pour chacune de ces classes. Ces métriques sont ensuite utilisées comme vecteurs de caractéristiques dans la classification. À l'étape d'entraînement, trois classificateurs sont construits : deux classificateurs basés sur les rôles, qui sont destinés à attribuer des rôles spécifiques aux classes et un classificateur basé sur les patrons, qui identifie le patron de type MVC appliqué dans une application. Dans l'étape de détection, les classificateurs entraînés ainsi que les vecteurs de caractéristiques construits au prétraitement sont employés afin de prédire le patron appliqué dans une application Android donnée. La suite du chapitre décrit en détail les différentes étapes de l'approche.

4.1 Prétraitement

Cette étape vise à représenter les classes significatives d'une application donnée avec des vecteurs de caractéristiques exploitables par les algorithmes de classification. Cette étape reçoit en entrée le code source d'une application Android, et produit en sortie des vecteurs de caractéristiques

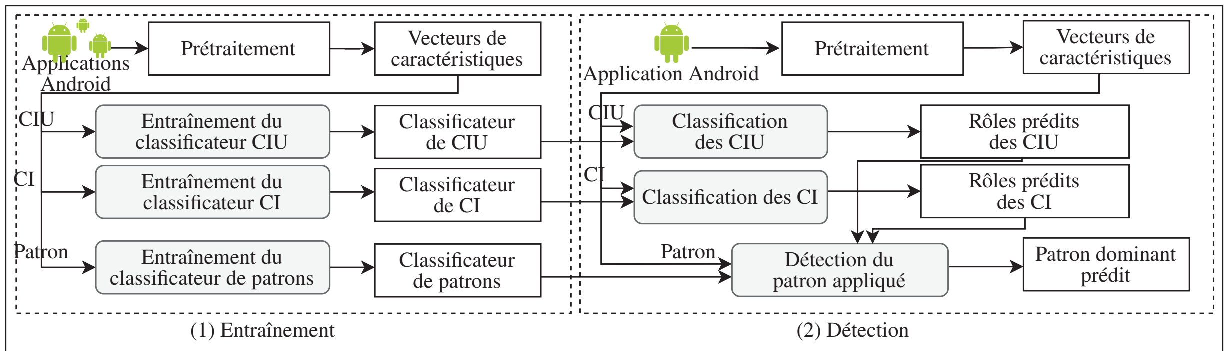

Figure 4.1 Aperçu de l'approche COACH

contenant des métriques du code source. Comme montré dans la figure 4.2, à partir du code source d'une application donnée, nous extrayons les classes significatives en utilisant des informations spécifiques à la plateforme Android. Ensuite, nous construisons des vecteurs de caractéristiques en calculant un ensemble de métriques de code sur les ensembles de classes extraits. Cette étape, à son tour, se décompose en deux sous-étapes suivantes :

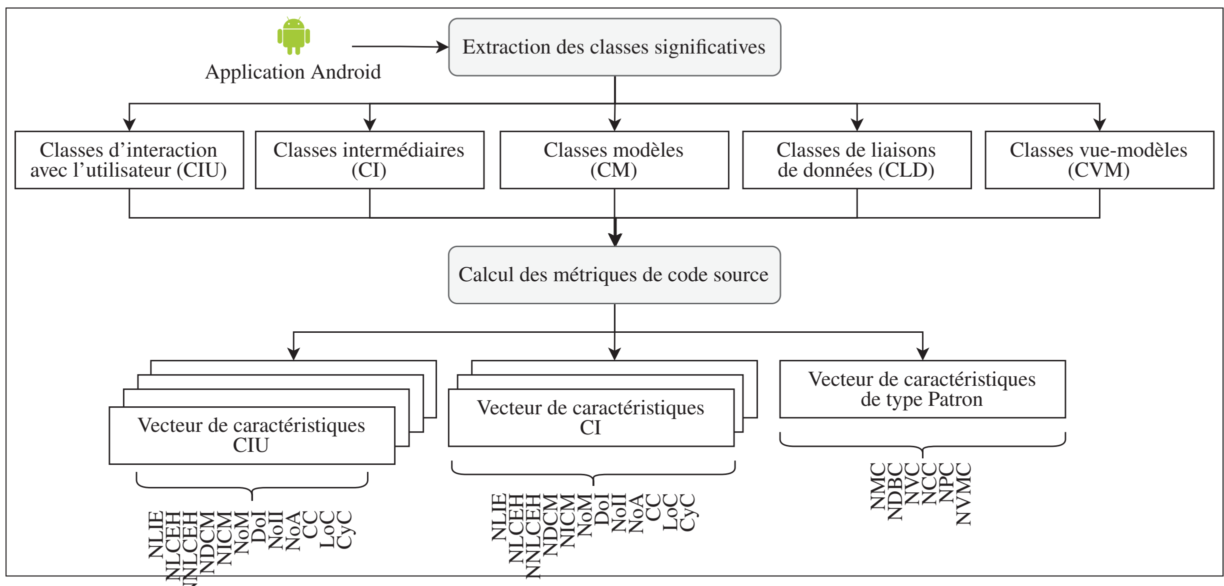

Figure 4.2 L'étape de prétraitement

4.1.1 Extraction des classes significatives

Cette sous-étape consiste à identifier les classes significatives existantes dans une application Android. Une classe significative désigne toute classe dans l'application qui participe à la différenciation entre les trois patrons étudiés en Android. Ces classes peuvent être identifiées à partir du code source sans aucune ambiguïté et constituent la première catégorisation des classes de l'application. Nous distinguons cinq catégories de classes significatives :

4.1.1.1 Classes modèles (CMs)

Cette catégorie regroupe les classes qui décrivent l'état du système, qui peut être persistant ou pas. Dans le contexte d'Android, les CMs peuvent être implémentées en tant que classes *POJO (Plain Old Java Object)*, ou construites à l'aide de bibliothèques ORMs, de bibliothèques de sérialisation ou de bibliothèques de base de données SQL. Pour identifier les CMs, nous nous appuyons sur le processus proposé par Daoudi *et al.* (2019). Ce dernier consiste à vérifier l'utilisation des bibliothèques les plus utilisées dans l'implémentation des CMs à travers une analyse statique du code source. À cette fin, une liste de 50 bibliothèques les plus utilisées, collectées à partir des sites web communautaires tels que *Android Arsenal* (Vladislav, 2019) et *Ultimate Android Reference* (Aritra, 2019) a été employée. Pour les CMs sous forme de classes *POJO*, nous considérons comme CM, toute classe ayant des attributs/constructeurs publics et uniquement des méthodes de type accesseur/mutateur.

L'identification des CMs est une étape nécessaire à la distinction entre le patron MVC et ses variantes. Certes, la principale différence entre ceux-ci réside dans la manière dont les interactions de l'utilisateur à travers les objets de la vue sont propagées aux objets du modèle. Par conséquent, la première étape à faire est d'identifier les CMs parmi les classes de l'application pour pouvoir par la suite suivre l'interaction de l'utilisateur avec elles. Il est à noter également que les objets de modèle représentés par les CMs sont implémentés de la même façon dans les trois patrons étudiés. Le même processus d'identification est donc applicable pour les trois patrons.

4.1.1.2 Classes d'interaction avec l'utilisateur (CIUs)

Les classes de cette catégorie se chargent de gérer les actions de l'utilisateur telles que les clics de boutons (*Button*) et l'ajout du texte dans les champs de type texte (*EditText*). En Android, les classes qui supportent ce type d'interactions sont les *activités*, les *fragments*, les *vues* et les *dialogues*. Ainsi, nous identifions les CIUs en recherchant toutes les classes qui ont au moins un parent dont le nom qualifié se termine par `.app.Activity`, `.app.Fragment`, `.app.Dialog` ou `.view.View`.

4.1.1.3 Classes de liaison de données (CLDs)

Les CLDs se chargent d'implémenter le mécanisme de liaison des données (*Data binding*), un concept clé du patron MVVM. Ce mécanisme permet la synchronisation et le découplage entre les composants de l'interface utilisateur et la logique métier. En Android, les CLDs sont générées automatiquement lors de la compilation et utilisées lors de l'exécution pour effectuer la liaison. Il est donc impossible de les repérer à partir du code source. Toutefois, l'utilisation de la liaison de données peut être tracée à partir des dépendances de l'application en inspectant l'utilisation de la bibliothèque *Data Binding Library* (Databinding, 2016). Une classe est considérée comme CLD lorsqu'elle utilise la classe `android.databinding.DataBindingUtil` à l'initialisation.

4.1.1.4 Classes vue-modèles (CVMs)

Telle qu'introduite par le patron MVVM, une CVM a la responsabilité de définir et d'implémenter les évènements d'entrée de l'utilisateur, de transformer les données du modèle et de les rendre prêtes à être affichées par la vue. En Android, après l'introduction de la bibliothèque *Jetpack* (Jetpack, 2017), l'implémentation du patron MVVM a été simplifiée à travers des composants de type `AndroidViewModel`. Ainsi, pour identifier une CVM, nous vérifions si la classe a au moins un parent dont le nom qualifié se termine par `androidx.lifecycle.ViewModel`.

4.1.1.5 Classes intermédiaires (CIs)

Cette catégorie regroupe les classes qui ne font pas partie des classes des catégories précédentes, et qui sont nécessaires à la différenciation entre les trois patrons étudiés. Nous considérons une classe comme CI lorsqu'elle :

1. n'étend aucune classe spécifique du cadre Android,
2. fait appel direct vers une classe CM ou CIU en faisant référence à ses attributs ou en appelant ses méthodes, ou
3. fait appel indirect vers une CM ou une CIU en passant par d'autres classes.

De cette façon, les classes CIs sont sélectionnées en tant que candidats potentiels pour faire partie du composant intermédiaire du patron appliqué, que ce soit un contrôleur, un présentateur ou un vue-modèle.

4.1.2 Calcul des métriques de code

À partir de l'ensemble des classes significatives extraites dans l'étape précédente (CIUs, CIs, CMs, CLDs et CVMs), nous produisons en sortie un ensemble de vecteurs de caractéristiques. Comme le montrent les figures 4.2 et 4.3, nous distinguons trois types de vecteurs de caractéristiques :

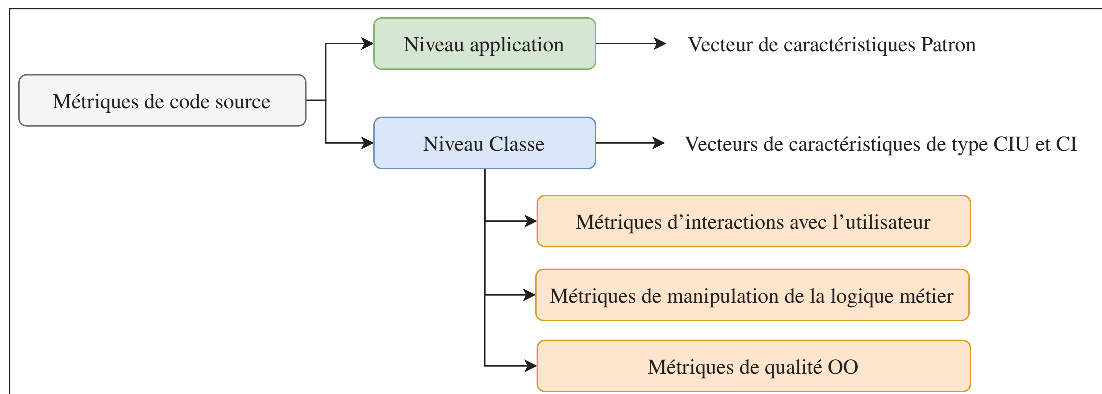

Figure 4.3 Classification des métriques de code source

4.1.2.1 Vecteur de caractéristiques de type CIU

Ce type de vecteurs de caractéristiques représente une classe de l'ensemble des CIUs. Il regroupe un ensemble de métriques de code source qui se concentrent sur les interactions avec l'utilisateur et la manière dont elles sont propagées aux CMs. En pratique, de telles métriques ne sont pas définies dans la littérature, surtout dans le contexte du développement d'Android. À cet effet, nous avons introduit un ensemble de métriques de code spécifiques à Android qui permettent de mesurer l'interaction avec l'utilisateur et la manipulation des CMs. Pour des raisons de clarté, nous les présenterons séparément dans la section 4.4. Nous avons également ajouté un certain nombre de métriques OO qui mesurent la complexité et les dépendances entre les classes. En effet, ce type de métrique peut être intéressant, car les patrons de type MVC permettent de réduire la complexité et le couplage entre les classes. De plus, l'ajout de métriques OO supplémentaires permet d'avoir une information complète sur les CIUs. Les métriques ajoutées sont présentées dans le tableau 4.1.

La liste finale des métriques de code constituant un vecteur de type CIU est la suivante :

$$V_{CIU} =< NLIE, NLCEH, NNLCEH, NDCM, NICM, NoM, DoI, NoII, NoA, CC, LoC, CyC >$$

Il convient également de noter que nous avons mené plusieurs expérimentations de la classification avec différents ensembles de métriques (avec et sans les métriques proposées) afin de valider leur pertinence pour notre approche. Nous en discuterons dans la section 4.6.

Tableau 4.1 Les métriques OO ajoutées

Métrique	Description
NoM	Nombre de méthodes de la classe
DoI	Profondeur d'héritage
NoII	Nombre d'interfaces implémentées par la classe
NoA	Nombre d'attributs de la classe
CC	Complexité de la classe
LoC	Cohésion entre les méthodes de la classe
CyC	Complexité cyclomatique

4.1.2.2 Vecteur de caractéristiques de type CI

Pareillement, ce vecteur de caractéristiques représente une classe de l'ensemble des CIs identifiées précédemment. Il comprend le même ensemble de métriques utilisées pour représenter les CIUs (c.-à-d. $V_{CI} =< NLIE, NLCEH, NNLCEH, NDCM, NICM, NoM, DoI, NoII, NoA, CC, LoC, CyC >$) En théorie, les métriques OO classiques de complexité et de couplage sont pertinentes dans ce cas puisque les CIs, qu'il s'agisse de contrôleurs, de présentateurs ou de vue-modèles, permettent de réduire la complexité d'une application en réduisant le couplage entre les CIUs et les CMs. Toutefois, seules, elles ne saisissent pas les notions d'interactions avec l'utilisateur et de manipulation des CMs qui sont essentielles à la différenciation entre les rôles possibles d'une CI. Il devient donc nécessaire d'utiliser les métriques proposées.

En pratique, nous avons validé cette liste des métriques en menant plusieurs expérimentations de classification avec différents ensembles de métriques. Les résultats obtenus montrent la pertinence de cette liste pour notre approche. Nous discuterons les détails dans la section 4.6.

4.1.2.3 Vecteur de caractéristiques de type patron

Ce vecteur de caractéristiques donne un aperçu sur les classes significatives existantes dans une application. Il agrège un ensemble de métriques de code qui sont calculées au niveau de l'application. Tableau 4.2 résume les métriques utilisées.

$$V_{Patron} =< NMC, NVMC, NDBC, NVC, NCC, NPC >$$

Tableau 4.2 Les métriques niveau application

Métrique	Description
NMC	Nombre de classes modèles
NVMC	Nombre de classes vue-modèle
NDBC	Nombre de classes de liaison de données
NVC	Nombre de classes vues
NCC	Nombre de classes contrôleurs
NPC	Nombre de classes présentateurs

Les trois premières métriques ($NMC, NVMC, NDBC$) sont calculées en utilisant l'ensemble des classes significatives ($CMs, CVMs, CLDs$) extraites dans la sous-étape précédente. Les autres métriques (NVC, NCC, NPC) sont calculées à partir des résultats de la classification basée sur les rôles, nous en discuterons dans les prochaines sections.

4.2 Étape 1 : entraînement

L'objectif de cette étape est de construire des classificateurs capables d'attribuer des rôles aux classes et de reconnaître le patron de type MVC utilisé dans une application Android. Comme le montre la figure 4.1, cette étape prend en entrée un ensemble d'applications annotées avec le patron appliqué et produit trois classificateurs en sortie : *classificateur des CIUs*, *classificateur des CIs* et *classificateur de patrons*. L'étape d'entraînement commence par l'application du prétraitement sur l'ensemble des applications annotées afin de générer les vecteurs de caractéristiques. Ces derniers sont ensuite utilisés pour construire les trois classificateurs comme suit :

4.2.1 Entraînement du classificateur des CIUs

À partir des vecteurs de caractéristiques de type CIU, nous construisons le classificateur des CIUs qui se charge d'attribuer un rôle à chaque classe CIU. Une CIU peut avoir un des rôles suivants :

- *Vue*, lorsque la classe se charge de l'affichage, gère les évènements de l'interface utilisateur et délègue la manipulation des CMs à d'autres classes ;
- *Contrôleur*, quand la classe se charge de l'affichage, gère les évènements de l'interface utilisateur et manipule les CMs en même temps ;
- *Aucun*, lorsque la classe se charge de l'affichage et gère les évènements de l'interface utilisateur sans aucune manipulation des CMs. Cela est nécessaire pour couvrir tous les cas possibles, par exemple, les activités affichant les informations «À propos» ou les applications sans CMs.

4.2.2 Entraînement du classificateur des CIs

Parallèlement, à partir de l'ensemble de vecteurs de caractéristiques de type CI calculés dans le prétraitement, nous construisons le classificateur des CIs, qui se charge d'attribuer un rôle à chaque classe CI. Une CI peut avoir un des rôles suivants :

- *Contrôleur*, lorsque la classe CI gère l'interaction entre les CIUs et les CMs sans être une classe CIU ;
- *Présentateur*, quand la classe CI gère l'interaction entre les CIUs et les CMs, mais en offrant plus de découplage que le rôle de contrôleur ;
- *Vue-Modèle*, lorsque la classe CI gère l'interaction entre les CIUs et CMs en utilisant le mécanisme de liaison de données ;
- *Aucun*, quand la classe CI ne respecte aucune définition des rôles précédents. Cela est également nécessaire afin de modéliser tous les cas possibles tels que les classes utilitaires.

4.2.3 Entraînement du classificateur de patrons

L'objectif de cette étape est de construire un classificateur qui peut assigner un patron à une application. Les patrons possibles sont : MVC, MVP, MVVM auxquels nous avons ajouté le patron *Aucun* pour indiquer que l'application n'applique aucun des trois patrons étudiés. Nous commençons l'entraînement en calculant les métriques manquantes dans les vecteurs de caractéristiques de type patron. Comme indiqué dans l'étape de prétraitement, ce type de vecteurs comprend les métriques (NVC, NCC, NPC) qui sont calculées en utilisant les résultats des classificateurs basés sur les rôles. Ensuite, nous construisons le classificateur de patrons à partir des vecteurs de caractéristiques de type patron complétés. Le résultat de cette étape est un classificateur de patrons entraîné.

4.3 Étape 2 : détection

L'étape de détection consiste à utiliser les classificateurs préalablement entraînés pour prédire le patron appliqué dans de nouvelles instances. Comme indiqué dans la figure 4.1, étant donné

une application Android, la détection commence par l'application du prétraitement sur le code source afin de générer les cinq ensembles de classes significatives (CIUs, CIs, CMs, CLDs et CVMs), et calculer les vecteurs de caractéristiques. Ensuite, les vecteurs de caractéristiques de type CIU sont injectés au classificateur de CIU entraîné, qui donne en sortie leurs rôles prédits. Le résultat est donc un ensemble de classes CIUs avec leurs rôles prédits. De la même manière, les vecteurs de caractéristiques de type CI sont injectés au classificateur de type CI. Le résultat est encore un ensemble de classes CIs avec leurs rôles prédits. À partir des résultats des deux classifications précédentes, le vecteur de caractéristiques de type patron est mis à jour en calculant les métriques de code manquantes (NVC, NCC, NPC). Le vecteur obtenu est ensuite injecté au classificateur de patron entraîné. Le résultat final est le patron de type MVC dominant appliqué dans l'application.

4.4 Métriques proposées

Comme nous l'avons énoncé précédemment, nous avons introduit de nouvelles métriques spécifiques à Android qui visent à mesurer l'interaction avec l'utilisateur et la manipulation de la logique métier, deux notions clés à la distinction entre les patrons de type MVC. Nous les décrivons dans les sections suivantes.

4.4.1 Métriques d'interaction avec l'utilisateur

En Android, l'interaction avec l'utilisateur est capturée via des évènements d'entrée, tels que le clic de boutons et l'édition des champs de type texte. Ces évènements sont généralement gérés à l'aide d'*écouteurs d'évènements* (*Event listener*) ou de *gestionnaires d'évènements* (*Event handler*) (Documentation-Android, 2022). À partir de là, nous introduisons les deux métriques suivantes :

4.4.1.1 Nombre d'évènements d'entrée basés sur des écouteurs (NLIE)

Description

Cette métrique mesure le nombre d'évènements d'entrée basés sur des écouteurs traités dans une classe donnée, tels que `onClick()` et `onFocusChange()`. Ces évènements sont déclenchés par des éléments de l'interface utilisateur et implémentés à l'aide du patron *Écouteur*.

Méthode de calcul

À travers une analyse statique du code source, nous identifions les évènements à base d'écouteurs en inspectant dans le code tout appel à une méthode d'enregistrement d'évènements (*Event registration*) effectuée par un élément de l'interface graphique. Figure 4.4 montre un exemple d'enregistrement de l'évènement `onClick()` d'un bouton. À cette étape, nous avons employé une liste des évènements basés sur des écouteurs, ainsi que leurs méthodes d'enregistrement correspondantes qui peuvent être déclenchés par des éléments de l'interface graphique dans la plateforme Android. La liste complète a été collectée à partir de la documentation officielle d'Android et contient 58 entrées (évènements et méthodes d'enregistrement). Nous considérons également le cas où l'enregistrement d'évènement est effectué au niveau des fichiers de mise en page (*Layout*) en vérifiant leur déclaration dans ces derniers.

4.4.1.2 Nombre de gestionnaires d'évènements non liés au cycle de vie (NNLCEH)

Description

Cette métrique mesure le nombre de gestionnaires d'évènements d'entrée non liés au cycle de vie [7], tels que `onKeyDown()` et `onKeyLongPress()`. Ces gestionnaires sont invoqués lorsque des évènements spécifiques sont déclenchés suite à une interaction avec l'utilisateur. Par

[7] En Android, il existe un autre type de gestionnaire d'évènements appelés gestionnaires d'évènements de cycle de vie. Ces derniers sont déclenchés automatiquement par le système et ne capturent pas l'interaction avec l'utilisateur.

```java
// Create an anonymous implementation of OnClickListener
private OnClickListener corkyListener = new OnClickListener() {
    public void onClick(View v) {
      // do something when the button is clicked
    }
};

protected void onCreate(Bundle savedValues) {
    ...
    // Capture our button from layout
    Button button = (Button)findViewById(R.id.corky);
    // Register the onClick listener with the implementation above
    button.setOnClickListener(corkyListener);
    ...
}
```

Figure 4.4 Exemple d'enregistrement d'évènement d'entrée, tiré de la Documentation-Android (2021b)

exemple, `onKeyDown()` est appelé lorsqu'un évènement de touche enfoncée s'est produit et `onKeyLongPress()` est déclenché lorsqu'une longue pression s'est produite. Ces évènements sont rarement utilisés, car ils ont été introduits afin de faciliter la personnalisation des composants de l'interface utilisateur (Documentation-Android, 2022).

Méthode de calcul

De la même manière, nous avons collecté, à partir de la documentation officielle d'Android, une liste de 42 gestionnaires d'évènements non liés au cycle de vie existants dans la plateforme Android. Ensuite, nous vérifions leur surcharge dans le code à travers une analyse statique. La liste complète des gestionnaires d'évènements considérés est disponible dans le paquet de réplication [8].

4.4.2 Métriques de manipulation de la logique métier

En Android, la logique métier représentée par les classes CMs peut être manipulée de plusieurs façons. Nous les catégorisons en trois métriques suivantes :

---

[8] Le paquet de réplication est disponible sur https://github.com/ChaimaChekhaba/COACH

4.4.2.1 Nombre de gestionnaires d'évènements du cycle de vie (NLCEH)

Description

Cette métrique calcule le nombre de gestionnaires d'évènements du cycle de vie tels que `onCreate()` et `onStop()`. Ce type de gestionnaires représente des points d'arrêt critiques pour les principaux composants de l'application, et peut être surchargé pour ajouter un comportement à exécuter lorsqu'ils sont déclenchés. En général, la surcharge de ces dernières fait appel à des opérations impliquant les objets du modèle. Par exemple, lorsque le gestionnaire d'évènements `onStop()` est appelé, le mécanisme de persistance de données est invoqué, manipulant ainsi les CMs.

Méthode de calcul

À travers une analyse statique du code source, nous inspectons la surcharge de tout gestionnaire d'évènements faisant partie de la liste des gestionnaires d'évènements du cycle de vie en Android. Cette dernière a été collectée à partir de la documentation officielle d'Android et contient 28 gestionnaires d'évènements du cycle de vie. Nous mettons à disposition la liste complète dans le paquet de réplication [9].

4.4.2.2 Nombre d'appels directs des CMs (NDCM)

Description

Cette métrique mesure le nombre d'appels directs effectués à partir d'une classe vers les classes CMs en faisant référence à leurs attributs ou en appelant leurs méthodes. Cela inclut toute déclaration manipulant une classe CM en lecture ou en modification. La manipulation en lecture se fait en récupérant la valeur d'un attribut statique ou d'instance, ou à travers des méthodes *getters()*. La manipulation en modification peut être l'initialisation d'une instance CM,

[9] Le paquet de réplication est disponible sur https://github.com/ChaimaChekhaba/COACH

la modification de la valeur d'un attribut statique ou d'instance, l'appel de méthodes *setters()*, de méthodes statiques, ou d'autres méthodes qui sont ni *getters()* ni *setters()*.

Méthode de calcul

Comme le montre l'algorithme 4.1, nous avons regroupé toutes les formes de déclarations possibles qui peuvent manipuler les classes CMs en deux cas :

1. *Cas 1 :* déclaration sous forme de $obj = object.method()$ ou $object.method()$, inclut la manipulation à travers l'appel de méthodes, que ce soit des méthodes *getters*, *setters*, statiques ou d'autres méthodes.
2. *Cas 2 :* déclaration sous forme de $object.attribut = method()$ ou $obj = object.attribut$, qui regroupe tout accès à un attribut (statique ou d'instance) en lecture ou en modification.

Ainsi, le calcul de cette métrique consiste à compter le nombre de déclarations dans toutes les méthodes de la classe concernée qui ont la forme d'un des deux cas précédents, et qui font référence à des classes CMs.

Algorithme 4.1 Pseudo code du calcul de la métrique NDCM

```
   Entrée : La liste des classes modèles CM, Classe C
   Sortie : Le nombre d'appels directs des CMs NDCM
1  NDCM = 0
2  Pour toute méthode m dans C faire
3  |   Pour toute déclaration stat dans m faire
4  |   |   Si stat est sous forme de object.method() ou obj = object.method() alors
5  |   |   |   CU ← Type(objet)                                  /* Cas 1 */
6  |   |   fin
7  |   |   Sinon stat est sous forme object.attribut = method() ou
   |   |    obj = object.attribut alors
8  |   |   |   CU ← Type(objet)                                  /* Cas 2 */
9  |   |   fin
10 |   |   Si CU est une classe dans CM alors
11 |   |   |   NDCM ← NDCM + 1
12 |   |   fin
13 |   fin
14 fin
```

4.4.2.3 Nombre d'appels indirects des CMs (NICM)

Description

Cette métrique mesure le nombre d'appels indirects faits par une classe vers les classes CMs en passant par d'autres classes.

Méthode de calcul

À travers un processus récursif, nous examinons le lien entre les classes utilisées par la classe concernée et les classes CMs. Algorithme 4.2 illustre le pseudo-code utilisé pour calculer cette métrique. Étant donné une classe C, nous identifions l'ensemble CC de toutes les classes dont C fait référence à leurs attributs ou appelle leurs méthodes. Ensuite, pour toute classe c dans CC, nous vérifions son lien avec les classes CMs en calculant la métrique NDCM. Dans le cas où la classe c ne manipule aucune classe CM, nous réappliquons le même processus sur toutes les classes qu'elle utilise.

Algorithme 4.2 Pseudo code du calcul de la métrique NICM

```
   Entrée : La liste des classes modèles CM, Classe C
   Sortie : Le nombre d'appels indirects des CMs NICM
1  NICM = 0
2  CC = ClasseUtiliseePar(C)
   /* Relation USES dans le diagramme de classe                    */
3  Pour toute classe c dans CC faire
4  |  Si NDCM(CM, c) > 0 alors
5  |  |  NICM ← NICM + 1
6  |  fin
7  |  Sinon alors
8  |  |  NICM ← NICM + NICM(CM, c)
9  |  fin
10 fin
```

4.5 Données d'apprentissage

L'ensemble de données d'apprentissage se compose de 69 applications Android libres collectées à partir du dépôt F-DROID (F-droid, 2021). Le processus de sélection consiste à collecter de manière aléatoire 69 applications [11] du dépôt F-DROID avec des liens valides vers GitHub. Par la suite, les applications collectées ont été analysées manuellement pour annoter leurs classes avec les rôles possibles et les classer comme implémentant soit MVC, MVP, MVVM ou aucun des patrons de type MVC. Pour extraire les métriques présentées dans la section 4.4, nous avons utilisé l'outil CUMIN [12], que nous avons implémenté en se basant sur l'implémentation de l'outil PAPRIKA (Hecht *et al.*, 2015). Tableau 4.3 montre la distribution des patrons de type MVC, telle que déterminée par notre annotation manuelle, dans l'ensemble de données d'apprentissage.

Tableau 4.3 Distribution des patrons de type MVC dans l'ensemble de données d'apprentissage

Patron	Entraînement	Test
MVC	20	6
MVP	10	2
MVVM	7	2
Aucun	18	4

Comme montré dans le tableau 4.3, le groupe d'applications le plus important est celui du MVC standard avec 26 applications, suivi des applications qui n'appliquent pas de patron (aucun) avec 22 applications. Les variantes MVP et MVVM représentent des parts beaucoup plus faibles, avec respectivement 11 et 9 applications.

Dans les expérimentations, nous avons réparti aléatoirement l'ensemble de données d'apprentissage en ensembles d'entraînement et de test (80 % pour l'entraînement et 20 % pour le test). Les tableaux 4.3 et 4.4 montrent le nombre d'instances dans les ensembles d'entraînement et

[11] Étant donné que l'annotation prend beaucoup de temps, nous avons dû limiter la taille de notre échantillon. Le nombre 69 nous fournit un échantillon représentatif de F-DROID, avec un niveau de confiance d'environ 95% et un intervalle de confiance de 10%.

[12] CUMIN est disponible dans https://github.com/ChaimaChekhaba/Cumin

de test après la répartition. L'ensemble d'entraînement contient 55 instances d'applications, dont les classes comprennent 724 CIUs et 506 CIs. L'ensemble de test, quant à lui, est composé de 14 instances d'applications avec 121 CIUs et 52 CIs. D'après le tableau 4.4, nous pouvons facilement remarquer le problème de *données déséquilibrées* dans les CIs. Les instances Aucun représentent plus de 86% des CI alors que les autres instances représentent moins de 14% du total. Ce cas particulier doit être traité durant l'étape de prétraitement et avant la classification, afin de limiter les risques de classification biaisée. Pour cela, nous avons appliqué la technique d'*échantillonnage de la classe minoritaire* (*Up-sample Minority Class*) (Kuhn & Johnson, 2013), qui consiste à échantillonner à nouveau la classe minoritaire (dans notre cas, les instances contrôleur, présentateur et vue-modèle) en dupliquant leurs occurrences dans l'ensemble de données d'apprentissage avec remplacement. Nous avons également pris en compte le poids des classes lors du calcul des métriques d'évaluation de la classification.

Tableau 4.4 Nombre de classes significatives dans l'ensemble de données d'apprentissage

Nombre de classes		Entraînement	Test
CIU	Vue	284	45
	Contrôleur	97	27
	Aucun	343	49
CI	Contrôleur	19	1
	Présentateur	32	4
	Vue-Modèle	16	6
	Aucun	439	41

4.6 Validation et expérimentations

Cette section rapportera les expérimentations réalisées afin d'évaluer notre approche, à savoir la validation des métriques utilisées dans la classification et la comparaison de notre approche avec l'état de l'art.

4.6.1 Questions de recherche

Pour évaluer notre approche, nous avons réalisé deux expérimentations. La première expérimentation vise à déterminer la pertinence des métriques utilisées dans la classification, plus précisément, la pertinence des métriques proposées pour la classification basée sur les rôles. À ce niveau, nous nous sommes concentrés sur les deux classifications basées sur les rôles, car elles emploient les métriques qui ont été introduites, et leurs résultats sont utilisés dans la classification de patron. La deuxième expérimentation vise à comparer l'approche COACH avec RIMAZ (Daoudi *et al.*, 2019), le seul outil existant dans la littérature, sur un ensemble d'applications Android.

Plus précisément, nos expérimentations visent à répondre aux deux questions de recherche suivantes :

1. **QR1 :** les métriques introduites sont-elles pertinentes pour la classification basée sur les rôles ?
2. **QR2 :** l'approche COACH basée sur la classification est-elle plus performante que l'approche RIMAZ basée sur des heuristiques pour la détection automatique des patrons de type MVC ?

Pour répondre à la première question de recherche (QR1), nous avons considéré trois groupes de métriques. Pour chaque groupe, nous avons effectué les deux classifications basées sur les rôles sur nos données d'apprentissage à l'aide des algorithmes de classification que nous avons sélectionnés. Les trois groupes de métriques sont les suivants :

- **Groupe 0 :** Métriques OO considérées par COACH ,
- **Groupe 1 :** Métriques d'interaction avec l'utilisateur et de manipulation de la logique métier,
- **Groupe 2 :** Métriques d'interaction avec l'utilisateur et de manipulation de la logique métier avec les métriques OO.

Pour répondre à la deuxième question de recherche (QR2), nous avons analysé les données de validation avec les deux outils COACH et RIMAZ. Les résultats obtenus sont discutés dans la section 4.7.

4.6.2 Algorithmes de classification

Il existe plusieurs algorithmes de classification dans la littérature. Choisir l'algorithme le plus approprié pour notre problème nécessite de les expérimenter sur nos données d'apprentissage. À cette fin, nous avons sélectionné un ensemble variant d'algorithmes de classification à partir de la bibliothèque *Scikit-learn* [13] de PYTHON, qui sont :

- Algorithmes basés sur les arbres : Arbre de décision, Forêt aléatoire, AdaBoost, et Bagging,
- Naive Bayes,
- Logistique simple,
- et Machines à vecteurs de support (SVM) avec différents noyaux : linéaire, sigmoïde et polynomiale.

Également, nous avons utilisé les paramètres par défaut des algorithmes de classification et l'avons évalué avec les trois métriques d'évaluation classiques : *précision (Pr)*, *rappel (R)* et *mesure F1 (F1)*.

4.6.3 Données de validation

En raison du manque des données de référence standards pour la validation, nous avons effectué l'évaluation sur un ensemble de 265 applications Android collectées à partir du GITHUB. Le processus de sélection consiste à récupérer les applications Android écrites en JAVA, où les développeurs indiquent le patron appliqué dans le fichier *Readme* ou la description de l'application, en utilisant l'API GITHUB. Par exemple, pour récupérer les applications Android avec le patron MVC, nous avons sélectionné tous les dépôts avec le tag «android» où JAVA est le langage principal, et la description ou le fichier *Readme* incluent «MVC». Nous avons limité les résultats aux 100 premières applications pour chaque patron en raison de contraintes de calcul. La liste obtenue est triée par ordre croissant de date de création afin de se concentrer davantage sur les applications complètement développées. Après avoir combiné les listes obtenues pour

[13] Scikit-learn Machine Learning in Python : https ://scikit-learn.org/stable/

chaque patron (MVC, MVP et MVVM) et supprimé les applications dupliquées, nous avons obtenu une liste de 265 applications.

Après avoir analysé l'ensemble de données de validation avec l'outil CUMIN pour calculer les métriques du code, nous avons remarqué que 52 de ces applications n'appliquent en réalité aucun patron en raison du petit nombre de classes (moins de 3). Pour cela, nous avons analysé ces applications manuellement pour s'assurer qu'aucun patron n'est appliqué. Nous avons obtenu un ensemble de données de validation contenant 48, 72 et 93 applications avec les patrons MVC, MVP et MVVM respectivement, en plus des 52 applications sans aucun patron. Tableau 4.5 illustre la distribution des patrons dans l'ensemble de données de validation.

Tableau 4.5 Distribution des patrons de type MVC dans l'ensemble de données de validation

Applications	MVC	MVP	MVVM	Aucun
265	48	72	93	52

4.7 Résultats et discussions

Dans cette section, nous discutons les résultats de nos expérimentations et répondons à nos questions de recherche.

4.7.1 QR1 : les métriques introduites sont-elles pertinentes pour la classification basée sur les rôles ?

Tableau 4.6 présente les résultats de la classification des CIUs pour chaque groupe de métriques. Tous les classificateurs ont obtenu de très faibles performances avec les métriques du groupe 0. Dans ce cas, le meilleur classificateur, qui est *Logistique simple*, se comporte comme un classificateur aléatoire avec une valeur F1 de 50%. Ces résultats indiquent que les métriques OO seules ne peuvent pas différencier correctement entre les différents rôles des classes CIUs. Avec les métriques du groupe 1, nous avons obtenu de meilleures performances pour tous les classificateurs comparés au groupe 0. Le meilleur classificateur, qui est le *SVM avec un noyau*

polynomial, a obtenu une valeur F1 de 57%. Cependant, ces résultats ne sont pas qualifiés de très satisfaisants ; ils indiquent que les métriques introduites sont pertinentes, mais pas suffisantes pour une classification efficace des CIUs. D'excellents résultats ont été obtenus uniquement en combinant les métriques OO et les métriques introduites. La plupart des classificateurs, à l'exception d'*AdaBoost*, de *Naïve Bayes* et de *SVM avec noyau sigmoïde*, ont obtenu des performances élevées. Les *Forêts aléatoires* ont présenté les meilleures performances avec 91% pour toutes les métriques d'évaluation, une précision de 91%, un rappel de 91% et une mesure F1 de 91%, qui sont des performances très élevées. Nous concluons que les métriques introduites sont nécessaires au même titre que les métriques OO pour différencier correctement entre les rôles des classes CIUs.

Tableau 4.6 Résultats de la classification des CIUs

Classificateur	Groupe 0			Groupe 1			Groupe 2		
	Pr	R	F1	Pr	R	F1	Pr	R	F1
Arbre de décision	0.35	0.35	0.35	0.50	0.50	0.50	0.84	0.84	0.84
Forêt aléatoire	0.35	0.35	0.35	0.57	0.51	0.51	**0.91**	**0.91**	**0.91**
Bagging	0.35	0.35	0.35	0.52	0.51	0.51	0.88	0.88	0.88
AdaBoost	0.25	0.38	0.29	0.58	0.53	0.54	0.52	0.51	0.45
Naïve Bayes	0.41	0.50	0.43	0.42	0.40	0.33	0.22	0.47	0.30
Logistique simple	**0.41**	**0.50**	**0.50**	0.55	0.55	0.53	0.68	0.68	0.68
SVM linéaire	0.35	0.22	0.23	0.59	0.54	0.56	0.71	0.70	0.70
SVM sigmoïde	0.19	0.20	0.16	0.49	0.34	0.38	0.53	0.46	0.47
SVM polynomial	0.39	0.30	0.33	**0.60**	**0.56**	**0.57**	0.79	0.73	0.72

Tableau 4.7 présente les résultats de la classification des CI pour chaque groupe de métriques. Contrairement à la classification CIU, les métriques du groupe 0 ont obtenu des performances relativement élevées. Les classificateurs basés sur les arbres ont atteint les meilleures performances, avec plus de 74% de valeur F1. Le classificateur *AdaBoost* est le plus performant avec une valeur F1 de 79%. De manière intéressante, des performances similaires ont été obtenues en utilisant les métriques du groupe 1. Tous les classificateurs ont été capables d'atteindre plus de 76% de valeur F1, à part *AdaBoost*, *SVM avec noyau linéaire* et S*VM avec noyau sigmoïde*. Les meilleures performances sont obtenues par les *Forêts aléatoires* avec plus de 88% pour toutes

les métriques d'évaluation. En combinant toutes les métriques de code, nous avons amélioré encore plus la classification des CI. Dans ce cas, le meilleur classificateur est le *Bagging*, qui a obtenu plus de 88% pour toutes les métriques d'évaluation. Ces résultats indiquent que les métriques proposées sont plus pertinentes que les métriques OO pour la classification des CI. En outre, leur utilisation seule est plus que suffisante pour une classification correcte des classes CIs. Ainsi, la réponse à notre première question de recherche est la suivante : **oui, les métriques introduites sont pertinentes et améliorent la classification basée sur les rôles.**

Tableau 4.7 Résultats de la classification des CIs

Classificateur	Groupe 0			Groupe 1			Groupe 2		
	Pr	R	F1	Pr	R	F1	Pr	R	F1
Arbre de décision	0.78	0.73	0.75	0.84	0.84	0.84	0.80	0.85	0.82
Forêt aléatoire	0.78	0.75	0.75	**0.88**	**0.89**	**0.88**	0.88	0.90	0.88
Bagging	0.78	0.71	0.74	0.86	0.84	0.85	**0.88**	**0.90**	**0.89**
AdaBoost	**0.77**	**0.81**	**0.79**	0.76	0.58	0.65	0.71	0.25	0.34
Naïve Bayes	0	0.06	0.01	0.83	0.73	0.77	0.81	0.68	0.74
Logistique simple	0.61	0.07	0.13	0.85	0.73	0.77	0.91	0.65	0.73
SVM linéaire	0.44	0.03	0.05	0.86	0.55	0.65	0.91	0.71	0.77
SVM sigmoïde	0	0.02	0	0	0.02	0.01	0.88	0.25	0.35
SVM polynomial	0.73	0.34	0.46	0.87	0.69	0.76	0.90	0.74	0.80

Les résultats de la classification de patrons sont présentés dans le tableau 4.8. Tous les classificateurs ont obtenu des performances très faibles, à l'exception de *Logistique simple* et du *SVM avec noyau linéaire*. Le classificateur *Logistique simple* a obtenu la meilleure performance avec une valeur F1 de 77%. D'après notre ensemble de données d'apprentissage, nous sommes en mesure de détecter correctement 77% des patrons de type MVC appliqués aux applications Android.

Tableau 4.8 Résultats de la classification de patrons

Classificateur	Précision	Rappel	Mesure F1
Arbre de décision	0.27	0.46	0.34
Forêt aléatoire	0.58	0.53	0.47
Bagging	0.27	0.46	0.34
AdaBoost	0.42	0.61	0.50
Naïve Bayes	0.25	0.31	0.26
Logistique simple	**0.79**	**0.77**	**0.77**
SVM linéaire	0.77	0.69	0.71
SVM sigmoïde	0.70	0.46	0.45
SVM polynomial	0.66	0.46	0.41

4.7.2 QR2 : l'approche COACH basée sur la classification est-elle plus performante que l'approche RIMAZ basée sur des heuristiques pour la détection automatique des patrons de type MVC ?

Tableaux 4.9 et 4.10 montrent les résultats obtenus après l'analyse de l'ensemble de données de validation avec COACH et RIMAZ. Pour le patron MVC, les deux outils ont exactement les mêmes performances, avec une valeur F1 de 68%. RIMAZ a correctement reconnu le MVC dans 42 applications, a mal classé trois applications comme étant MVP, une application comme n'appliquant aucun patron et a échoué à analyser deux applications (TIMEOUT). De son côté, COACH a correctement identifié le MVC dans 44 applications et a classé à tort deux applications en tant que MVP et deux autres en tant qu'applications sans patron (aucun). Quant au patron MVP, COACH a obtenu de meilleurs résultats comparés au RIMAZ avec une valeur F1 de 56% ; 33 applications ont été correctement classées, 16 applications classées à tort en MVC, une application en MVVM et 22 applications sans patron (aucun). Tandis que RIMAZ a réussi de reconnaître le MVP dans sept applications seulement et a mal classé 11 applications en tant que MVC, 22 en tant qu'applications sans patron (aucun), et a échoué à analyser 32 applications (TIMEOUT). RIMAZ n'a pas réussi à identifier le patron MVVM dans l'ensemble de données de validation, contrairement à COACH qui a atteint une valeur F1 de 67%, soit 48 applications correctement classées, et 16 applications confondues en MVC, 6 applications en MVP et 24 applications comme n'appliquant aucun patron. Pour les applications sans patron (aucun), les

deux outils ont obtenu des performances très proches, avec un léger écart pour RIMAZ, qui a obtenu une valeur F1 de 62%. L'outil COACH a correctement reconnu l'absence de patron dans 44 applications et a échoué dans huit applications (quatre classées en MVC et quatre en MVP). En revanche, RIMAZ a classé 50 applications avec succès et a manqué deux applications en MVC.

Tableau 4.9 Les résultats obtenus par COACH et RIMAZ en termes de précision, de rappel et de F1

Patron	COACH			RIMAZ		
	Précision	Rappel	Mesure F1	Précision	Rappel	Mesure F1
MVC	0.55	0.91	**0.68**	0.55	0.91	**0.68**
MVP	0.73	0.45	**0.56**	0.33	0.17	0.22
MVVM	0.97	0.51	**0.67**	0	0	0
Aucun	0.47	0.84	**0.61**	0.52	0.96	**0.62**

Tableau 4.10 Les résultats obtenus par COACH et RIMAZ en termes de nombre d'applications

Vrais patrons	Patrons prédits							
	MVC		MVP		MVVM		Aucun	
	COACH	RIMAZ	COACH	RIMAZ	COACH	RIMAZ	COACH	RIMAZ
MVC	44	42	2	3	0	0	2	1
MVP	16	11	33	7	1	0	22	22
MVVM	16	21	6	11	48	0	23	23
Aucun	4	2	4	0	0	0	44	50
TIMEOUT	-	2	-	32	-	38	-	0

À partir des résultats obtenus, nous avons tiré les conclusions suivantes :

1. Rimaz n'est pas toujours en mesure d'analyser les applications Android et reconnaître le patron appliqué, étant donné qu'il n'a pas réussi à analyser 72 applications de l'ensemble de données de validation en raison de l'erreur TIMEOUT. Ce problème n'est pas survenu avec notre approche Coach.
2. Coach et Rimaz procèdent de manière similaire pour la détection des patrons de type MVC. Les métriques d'interaction avec l'utilisateur et de manipulation de la logique métier introduites sont très comparables aux heuristiques utilisées par Rimaz dans l'analyse. Cela explique les résultats semblables obtenus dans la détection du patron MVC et l'absence de patron (Aucun).
3. Coach surpasse Rimaz dans la détection du patron MVP. Cela s'explique par le fait que nous considérons les classes Vue (*View*) qui étendent les classes du cadre d'applications Android en tant que classes d'interaction avec l'utilisateur (CIU). En revanche, Rimaz les traite comme des classes auxiliaires, ce qui conduit à la possibilité de les considérer comme des classes présentateurs. En outre, toutes les applications appliquant le patron MVP dans l'ensemble de données d'apprentissage déclarent les présentateurs sous forme de classes personnalisées indépendantes du cadre d'applications Android.
4. Coach peut détecter le patron MVVM dans les applications Android, contrairement à l'outil Rimaz. Cela est dû au fait que toutes les applications avec le patron MVVM dans nos ensembles de données (apprentissage et validation) ont au moins une classe `AndroidViewModel` qui s'étend du cadre d'applications Android. En outre, nous considérons le nombre des classes vue-modèle dans notre classification de patron (NVMC). En revanche, Rimaz inspecte les classes de liaison de données qui sont générées à la compilation.

Bien que l'approche Coach ait montré de meilleures performances comparées à Rimaz, nous devons souligner ses limites qui doivent être abordées de manière appropriée dans des travaux ultérieurs. Nous en discutons ainsi :

- L’identification des classes modèles (CMs) se fait via des heuristiques en inspectant l’utilisation des bibliothèques les plus populaires dans l’implémentation des CMs. Cela permet d’identifier un grand nombre d’implémentations de CMs, mais pas toutes les implémentations existantes.
- Les métriques utilisées dans la classification basée sur les rôles ne capturent pas la totalité des sémantiques des rôles dans les patrons de type MVC. Par exemple, pour le rôle Vue, aucune métrique utilisée ne capture la logique de présentation. Nous nous sommes donc appuyés sur les métriques OO pour les caractériser. De nouvelles métriques devraient être élaborées pour ces rôles à part.
- La définition des rôles dans la tâche de classification des classes d’interaction (CIUs) n’est pas précise. En fait, certains rôles réunissent plusieurs sémantiques dans les patrons de type MVC. En particulier, le rôle Contrôleur qui combine la logique de présentation, l’interaction avec l’utilisateur et la manipulation des CMs. Cela n’a pas d’impact sur le processus de détection global tant que cette définition a été respectée dans l’annotation manuelle. Toutefois, elle ne modélise pas le contexte réel, car les concepts de logique de présentation, l’interaction avec l’utilisateur et la manipulation des CMs portent des sémantiques différentes. Par conséquent, il est approprié de diviser les rôles définis en rôles atomiques.

Dans l’ensemble, Coach est en mesure d’améliorer la détection des patrons de type MVC dans les applications Android. Même avec un petit ensemble de données d’apprentissage, nous avons pu obtenir de meilleures performances en comparaison avec l’état de l’art. Avec plus d’applications de différentes implémentations, nous pouvons encore améliorer la détection. En conclusion, la réponse à notre deuxième question de recherche est : **notre approche basée sur la classification est plus performante que l’approche Rimaz basée sur des heuristiques.**

4.8 Conclusion

Dans ce chapitre, nous avons présenté notre approche Coach qui vise à identifier les patrons de type MVC dans les applications Android à travers la classification traditionnelle. Nous avons

introduit de nouvelles métriques spécifiques à Android permettant de quantifier l'interaction de l'utilisateur et la manipulation de la logique métier. Nous avons également exposé notre ensemble d'apprentissage contenant 69 applications Android libres annotées manuellement. Nous avons évalué COACH sur un ensemble de 265 applications Android, et comparé les résultats obtenus avec l'outil RIMAZ. Nos résultats suggèrent que COACH surpasse l'état de l'art dans la détection de tous les patrons de type MVC.

Le prochain chapitre décrit notre deuxième contribution MLCOACH. Cette dernière peut être considérée comme une extension de l'approche COACH pour la détection de combinaisons de patrons de type MVC en Android en exploitant une autre technique d'apprentissage automatique : la classification multi-label.

CHAPITRE 5

DÉTECTION DES PATRONS DE TYPE MVC DANS LES APPLICATIONS ANDROID À L'AIDE DES TECHNIQUES DE CLASSIFICATION MULTI-LABEL : L'APPROCHE MLCOACH

Dans ce chapitre, nous présentons notre deuxième contribution principale : l'approche MLCOACH (*Multi Label Classification Oriented Approach for ArChitectural pattern detection*). Cette dernière permet de détecter la combinaison de patrons de type MVC dans les applications Android grâce à la classification multi-label. L'approche MLCOACH utilise les techniques d'apprentissage profond et exploite de multiples sources d'informations extraites à partir du code source d'applications Android. Nous présentons également une analyse qualitative visant à comprendre les facteurs qui poussent les développeurs à mélanger différentes variantes du MVC au sein d'une même application. À notre connaissance, notre étude est la première dans la littérature à s'intéresser à l'analyse de mélange de patrons de type MVC dans les applications Android. Elle est également la première à enquêter sur les facteurs qui conduisent à de telles implémentations.

L'utilisation de la classification multi-label est au cœur de notre approche. Elle nous a permis de formaliser correctement notre deuxième problématique. En pratique, une application Android peut appliquer un patron unique, mélanger plusieurs patrons à la fois, ou n'appliquer aucun des trois patrons. Cela correspond à un problème classique de classification multi-label, où une instance (application) peut avoir un seul label, plusieurs labels ou aucun. Il en va de même au niveau des rôles. En effet, l'implémentation des patrons de type MVC dans les applications Android peut se manifester par l'implémentation de plusieurs rôles dans une même classe. Comme nous l'avons vu précédemment, l'une des implémentations les plus courantes utilisées par les développeurs Android consiste à considérer les *activités* comme vues et contrôleurs en même temps. Cela dit, une classe peut avoir plusieurs rôles simultanément. Dans ce cas, la formalisation la plus précise est de définir ce problème en tant que problème de classification multi-label, où une classe peut avoir un seul rôle, plusieurs rôles ou aucun.

Illustrée dans la figure 5.1, l'approche MLCOACH se décline en deux étapes : l'entraînement et la détection. L'étape de l'entraînement consiste à construire deux types de classificateurs : deux classificateurs basés sur les rôles qui attribuent des rôles aux classes, et un classificateur de patrons qui assigne le(s) patron(s) approprié(s) aux applications. À la détection, les trois classificateurs entraînés sont employés afin de prédire le(s) patron(s) appliqué(s) pour une application donnée. La suite du chapitre décrira en détail les différentes étapes de l'approche.

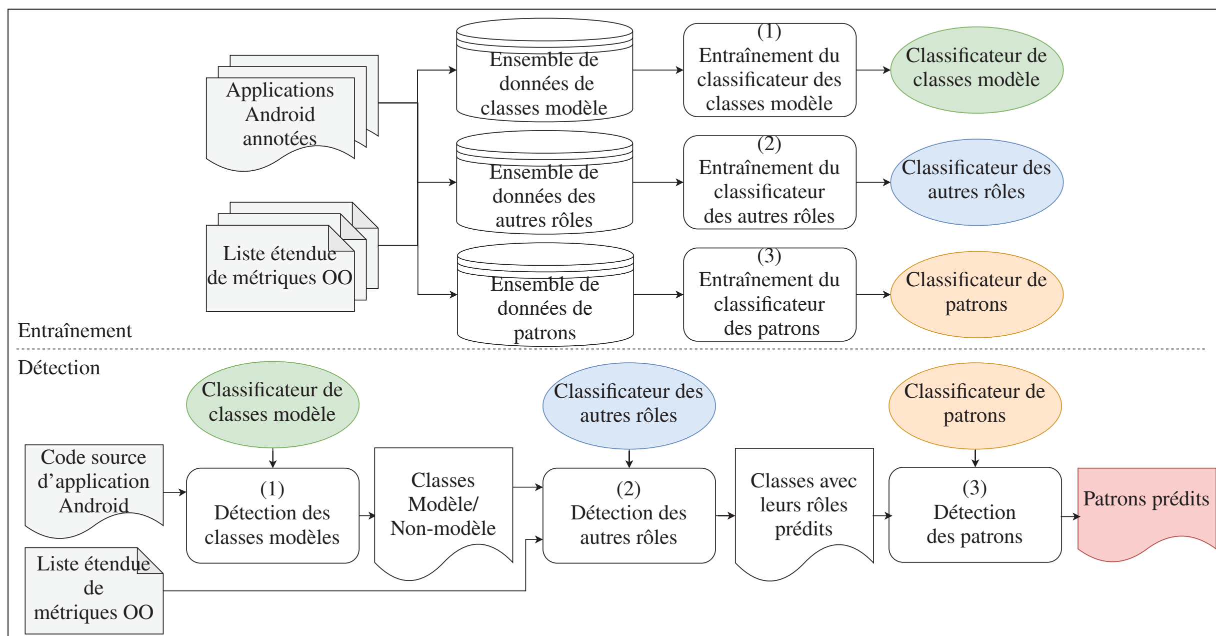

Figure 5.1 Aperçu de l'approche MLCOACH

5.1 Étape 1 : entraînement

Cette étape vise à construire trois classificateurs capables d'assigner des rôles aux classes et de patrons aux applications. Comme le montre la figure 5.1, l'entraînement prend en entrée un ensemble d'applications Android annotées manuellement, ainsi qu'une liste étendue des métriques OO correspondantes, et produit en sortie les trois classificateurs suivants :

1. *Classificateur des classes modèles*, qui se charge d'attribuer le rôle modèle aux classes ;
2. *Classificateur des rôles*, qui assigne des rôles autres que modèle aux classes (c.-à-d., Vue, Contrôleur, Présentateur et Vue-modèle) ; et

3. *Classificateur de patrons*, qui attribue des patrons aux applications (c.-à-d., MVC, MVP, MVVM et Aucun).

À partir de l'ensemble d'applications annotées, nous commençons l'entraînement par la construction de trois ensembles de données. Chaque ensemble étant utilisé pour une tâche de classification spécifique. Les ensembles de données sont les suivants :

- **Ensemble de données des classes modèles (DCM)**, contenant toutes les classes annotées en tant que modèle dans l'ensemble des applications initial, ainsi qu'un ensemble de classes ayant des rôles autres que modèle (c.-à-d., Vue, Contrôleur, Présentateur, Vue-modèle ou Aucun). Ces dernières ont été choisies au hasard afin d'équilibrer les données d'apprentissage.
- **Ensemble de données des autres rôles (DAR)**, qui inclut les classes ayant des rôles autres que modèle (c.-à-d., Vue, Contrôleur, Présentateur, Vue-modèle ou Aucun) dans l'ensemble d'applications initial. Certaines classes de cet ensemble font également partie de l'ensemble de données DCM, qui ont été utilisées afin de rendre l'ensemble de données DCM équilibré. Cela n'affecte pas notre approche tant que les deux ensembles de données sont utilisés différemment dans chaque tâche de classification.
- **Ensemble de données des patrons (DP)**, qui comprend toutes les applications annotées avec leurs patrons de type MVC correspondants.

Ces trois ensembles de données sont ensuite utilisés pour entraîner les trois classificateurs de l'approche.

5.1.1 Entraînement du classificateur des classes modèles

5.1.1.1 Entrée

Le code source des classes de l'ensemble DCM.

5.1.1.2 Sortie

Classificateur de classes modèles entraîné.

5.1.1.3 Objectif

L'objectif de cette tâche de classification est de différencier les classes modèles des classes ayant d'autres rôles. Rappelons que la principale différence entre les patrons de type MVC réside dans la manière dont l'interaction de l'utilisateur est propagée des classes vues aux classes modèles. Par conséquent, l'identification des classes modèles est nécessaire pour évaluer comment elles sont manipulées par les classes avec d'autres rôles. Les classes modèles identifiées seront utilisées dans les étapes suivantes pour calculer les métriques de code liées aux manipulations du modèle.

5.1.1.4 Méthodologie

Selon la définition standard des patrons MVC, MVP et MVVM et notre analyse manuelle de l'ensemble des applications annotées, la sémantique du rôle modèle est similaire parmi ces trois patrons. De plus, les classes modèles sont généralement implémentées à l'aide des ORMs, de sérialisations ou de bibliothèques de bases de données SQL (SQLDB) (Daoudi *et al.*, 2019). Pour cette tâche de classification, nous avons évalué deux stratégies différentes et sélectionné l'implémentation la plus efficace en termes de précision. Les stratégies qui ont été considérées sont les suivantes :

- **Stratégie basée sur des heuristiques :** dans cette stratégie, nous nous appuyons sur le processus d'identification des classes modèles utilisé dans notre première approche COACH. Ce dernier consiste à vérifier l'utilisation des implémentations les plus populaires des classes modèles, que ce soit des bibliothèques ou des classes POJO, à l'aide de l'analyse statique du code source.
- **Stratégie basée sur l'apprentissage profond :** nous exploitons les représentations distribuées du code (*code embeddings*) à partir de l'arbre syntaxique abstrait (AST) pour extraire les

propriétés sémantiques et les informations textuelles du code de chaque classe. La motivation derrière l'utilisation de cette approche est que les représentations distribuées des mots ont joué un rôle clé dans différentes tâches de traitement du langage naturel (NLP) (Yu, Wang, Lai & Zhang, 2017; Manning, Raghavan & Schütze, 2010; Chen, Zhang & Zhang, 2014; Zhou, Xie, He, Zhao & Hu, 2016). En outre, les classes modèles ont tendance à exploiter un vocabulaire similaire provenant de bibliothèques communes. Dans la suite de la section, nous nous concentrons sur cette stratégie, car elle permet d'obtenir les meilleurs résultats selon nos expérimentations, nous discuterons les détails dans la section 5.3.

Comme le montre la figure 5.2, nous effectuons l'entraînement de notre classificateur via un pipeline qui ressemble à un processus d'entraînement classique des tâches NLP. Tout d'abord, étant donné le code source d'une classe, nous effectuons plusieurs opérations de prétraitement afin de préparer les données et les transformer de texte en vecteurs numériques qui peuvent être traités par un réseau de neurones profond. Ces opérations comprennent :

1. La tokenisation pour déterminer le vocabulaire de l'ensemble de données,
2. L'indexation des mots de façon à ce que l'indice soit inversement proportionnel à la fréquence d'occurrence du mot, par exemple, le mot le plus fréquent aurait l'indice 1.

Ensuite, le code source est codé en utilisant le dictionnaire de vocabulaire en remplaçant chaque mot par son indice correspondant. Après, nous utilisons l'apprentissage par transfert en projetant les mots encodés dans un espace vectoriel de 128 dimensions à partir du modèle des représentations distribuées. Enfin, les vecteurs résultants sont injectés comme couche d'entrée du réseau de neurones profond.

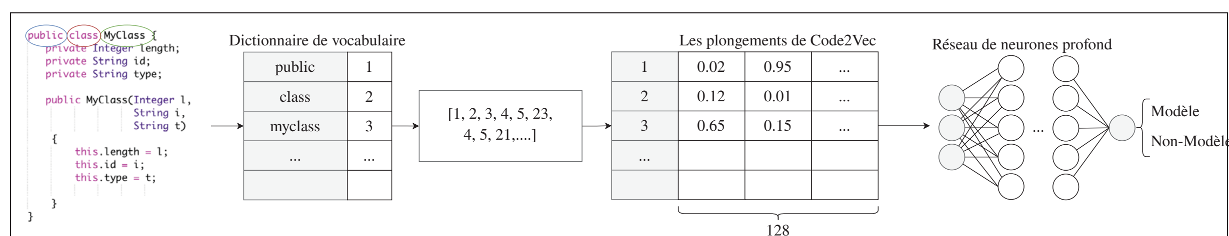

Figure 5.2 Aperçu de la tâche de classification des classes modèles

Pour l'implémentation, nous avons utilisé le modèle *Code2Vec* (Alon, Zilberstein, Levy & Yahav, 2019) qui a été entraîné sur un ensemble de données d'environ 16 millions d'exemples provenant de 9 500 projets JAVA les mieux notés sur GITHUB (Alon, Brody, Levy & Yahav, 2018a). Ensuite, nous avons introduit les vecteurs de représentation dans un réseau de neurones profond composé de différentes architectures que nous avons expérimentées séparément. Ces architectures comprennent des couches de perceptron multicouche (MLP) et de réseau de neurones convolutif (CNN). Chaque architecture offre une stratégie différente de modélisation et de résolution du problème évoqué. D'une part, le MLP permet de construire une architecture entièrement connectée de couches denses qui sont plus efficaces pour résoudre les problèmes non linéaires, mais une architecture profonde nécessite beaucoup de données en raison du grand nombre de paramètres. D'autre part, les CNN sont plus efficaces pour capturer la présence de certains patrons dans les données et extraire la relation sémantique entre les occurrences.

5.1.2 Entraînement du classificateur des autres rôles

5.1.2.1 Entrée

Le code source des classes de l'ensemble de données DAR.

5.1.2.2 Sortie

Classificateur des autres rôles entraîné.

5.1.2.3 Objectif

Cette étape vise à construire un classificateur capable d'attribuer de(s) rôle(s) aux classes (autres que le rôle modèle prédit par le classificateur des classes modèles). Une classe peut avoir un ou plusieurs des rôles suivants :

- *Vue*, lorsque la classe gère les fonctions d'affichage et met à jour l'interface utilisateur graphique (GUI) en étendant une des classes suivantes du cadre d'Android : `.app.Activity`, `.app.Fragment`, `.app.Dialog`, ou `.view.View` ;
- *Contrôleur*, quand la classe se charge de la gestion des évènements d'entrée déclenchés par la vue, met à jour l'interface graphique de la vue et manipule les classes du modèle ;
- *Présentateur,* lorsque la classe utilise des méthodes qui manipulent les classes du modèle et met à jour l'interface graphique de la vue, tout en référençant les classes de vues par le biais d'interfaces ou de classes génériques ;
- *Vue-modèle*, quand la classe déclare des méthodes manipulant les classes du modèle et implémentant les mises à jour automatiques des données affichées dans les classes de vue par le biais des classes `LiveData` ou `Observable`. Il inclut également le cas de la liaison de données couverte au niveau des fichiers de mise en page XML associés aux classes des vues ;
- *Aucun*, lorsqu'il s'agit d'une classe qui ne correspond à aucun rôle dans l'implémentation des patrons de type MVC.

5.1.2.4 Méthodologie

Figure 5.3 illustre le déroulement de cette tâche de classification. Au départ, à partir du code source des classes de l'ensemble de données DAR, nous appliquons un prétraitement, qui comprend le calcul d'une liste de métriques de code source pertinentes à la différenciation entre les rôles possibles dans les patrons de type MVC. Comme nous l'avons vu précédemment, les métriques OO classiques ne permettent pas de capturer la sémantique des différents rôles. À cet effet, nous nous sommes appuyés sur les métriques d'interaction avec l'utilisateur et de manipulation de la logique métier proposées dans la section 4.4 de l'approche Coach, qui permettent une séparation plus explicite entre les différents rôles. Nous avons également introduit d'autres métriques de code source telles que `NGUIC`, `NDS` et `NIE_XML` qui ciblent principalement les opérations d'affichage et la gestion des évènements d'entrée à partir des fichiers de mise en page. Cela est nécessaire afin de capturer la sémantique du rôle Vue. Tableau 5.1 présente la liste des métriques introduites avec une brève justification de leur utilisation. À cette liste, nous avons

ajouté 52 métriques OO classiques qui ont été calculées à l'aide de l'outil Metrics-Reloaded (Jet-Brains-Marketplace, 2021). Ensuite, les métriques de code calculées sont injectées au réseau de neurones artificiels (ANN) comme vecteurs de caractéristiques afin de différencier les différents rôles.

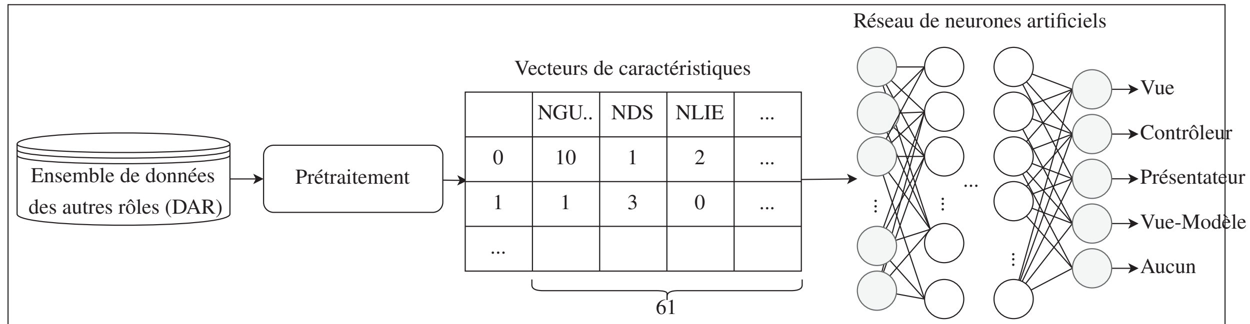

Figure 5.3 Aperçu de la tâche de classification des autres rôles

Comme le montre la figure 5.3, l'architecture du réseau de neurones artificiels utilisé est constituée de plusieurs couches denses, allant de la couche d'entrée avec 61 neurones à la couche de sortie avec cinq neurones. Pour choisir le nombre de neurones dans les couches cachées, nous nous sommes appuyés sur une heuristique largement utilisée (Heaton-Research, 2021) puisqu'il n'existe pas de stratégie standard permettant de déterminer ces hyper-paramètres (nombre de couches cachées et nombre de neurones par couche).

Pour réaliser la classification multi-label, nous avons utilisé la fonction d'activation `sigmoïde` dans la couche de sortie et `binary_crossentropy` comme fonction de perte. Il est à noter qu'avec cette configuration, nous avons appliqué plusieurs classifications binaires, chacune d'entre elles permettant de classer un seul rôle. Pour les couches cachées, nous avons utilisé la fonction d'activation `relu`. Le choix d'une telle architecture a été motivé par la capacité des réseaux de neurones artificiels à bien séparer les données à haute dimension. De plus, après plusieurs expérimentations, nous avons opté pour cette architecture, car elle montrait les meilleures performances en termes de métriques d'évaluation (voir la sous-section 5.3.3).

Tableau 5.1 Métriques de code source introduites

Métrique	Description
NGUIC	Nombre de composants d'interface graphique (tels que `Button`, `EditText` et `TextView`) déclarés dans les fichiers XML associés à la classe à travers la méthode `setContentView()`. Cette métrique permet de caractériser le rôle Vue, puisque les classes de la vue sont chargées de contenir et de gérer les composants de l'interface graphique.
NDS	Nombre d'opérations d'affichage dans la classe, telles que `setText()` et `setBackgroundColor()`. Elle fournit une mesure approximative des mises à jour et des opérations d'affichage déclarées par la classe.
NIE_XML	Nombre d'évènements d'entrée déclarés dans les fichiers de mise en page XML (tels que `android:onClick`. En Android, la gestion des évènements d'entrée peut également être réalisée par le biais du fichier de mise en page associé à la classe. Cela va compléter les métriques ciblant l'interaction avec l'utilisateur.
extendsAndroidViewModel	vrai si la classe s'étend de la classe `AndroidViewModel` fournie par le cadre d'applications Android. Cet indicateur booléen peut être intéressant à la reconnaissance du rôle Vue-modèle puisque toutes les applications appliquant le patron MVVM utilisées dans l'entraînement utilisent cette classe en particulier.

5.1.3 Entraînement du classificateur de patrons

5.1.3.1 Entrée

L'ensemble de données de patrons (DP).

5.1.3.2 Sortie

Classificateur de patrons entraîné.

5.1.3.3 Objectif

Cette tâche de classification vise à créer un classificateur capable d'assigner de(s) patron(s) aux applications. Les patrons possibles sont : MVC, MVP, MVVM auxquels nous avons ajouté le patron Aucun pour indiquer que l'application n'applique aucun des trois patrons étudiés. À ce stade, le classificateur doit être en mesure d'identifier tous les patrons utilisés dans l'application et de déterminer les scores de similarité associés à chacun d'entre eux.

5.1.3.4 Méthodologie

Figure 5.4 montre le déroulement de cette tâche de classification. Au départ, à partir de l'ensemble de données DP, nous appliquons un prétraitement. Ce dernier consiste à calculer une liste de métriques de code source au niveau de l'application, qui sont ensuite utilisées comme vecteurs de caractéristiques. Cette liste regroupe les valeurs moyennes [14] de quatre métriques OO calculées pour chaque ensemble de classes avec un rôle particulier (c.-à-d., métrique OO moyenne /rôle). Par exemple, pour le rôle modèle, nous avons calculé la moyenne de la complexité de toutes les classes avec ce rôle. Les quatre métriques OO considérées sont : la complexité (CC), la cohésion entre les méthodes de la classe (LoC), la complexité cyclomatique (CyC) et le couplage entre les objets (Co). De la même façon, nous avons calculé les valeurs moyennes des métriques OO sélectionnées pour les autres rôles, c.-à-d., Vue, Contrôleur, Présentateur et Vue-modèle. À cette liste, nous avons ajouté d'autres métriques qui donnent un aperçu sur le nombre de classes ayant un rôle spécifique dans l'application, comme le nombre de classes de modèle (NMC) et le nombre de classes de vue-modèle (NVMC). Ainsi, chaque rôle est représenté par cinq métriques :

$$Role =< CC\ Moyenne, LoC\ Moyenne, CyC\ Moyenne, Co\ Moyenne,$$
$$Nombre\ De\ Classes >$$

La liste complète des métriques est fournie dans le tableau 5.2. Enfin, les métriques calculées

---

[14] Pour simplifier le texte, nous utilisons la moyenne pour désigner l'écart type, et non la moyenne d'une série statistique.

sont injectées au classificateur de patrons comme vecteurs de caractéristiques afin de distinguer entre les trois patrons de type MVC.

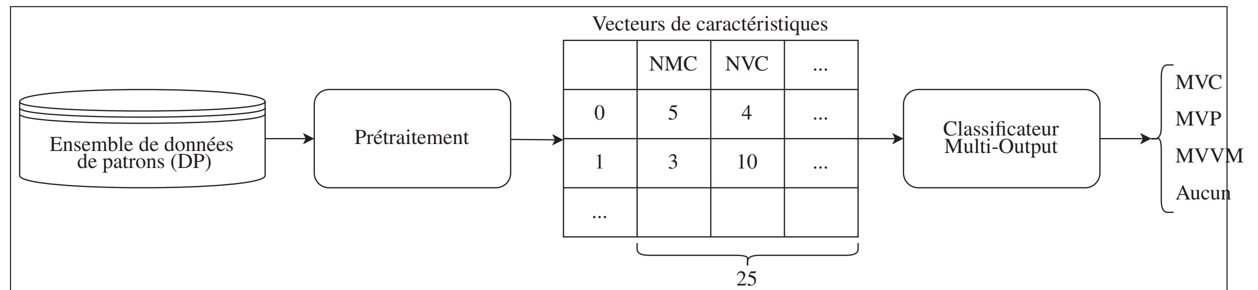

Figure 5.4 Aperçu de la tâche de classification de patrons

Pour implémenter la classification multi-label, nous avons effectué plusieurs classifications binaires sans utiliser les réseaux de neurones artificiels. Ceci est dû à la petite taille de l'ensemble DP (69 applications seulement) qui ne suffit pas pour entraîner un réseau de neurones artificiels, car ils nécessitent beaucoup de données. À cette fin, nous avons utilisé la fonction `MultiOutputClassifier` de la bibliothèque Scikit-learn [15] avec `LogisticRegression` comme classificateur binaire de base. Nous avons opté pour ce choix après avoir expérimenté plusieurs algorithmes de classification traditionnels.

5.2 Étape 2 : détection

5.2.1 Entrée

Le code source d'une application Android et les métriques OO correspondantes.

5.2.2 Sortie

Les patron(s) appliqué(s) dans l'application.

[15] https ://scikit-learn.org/stable/

Tableau 5.2 Métriques utilisées pour la classification de patrons

Métrique	Description
NCM	Nombre des classes modèles
CCM	Complexité moyenne des classes modèles
CNCM	Complexité moyenne de type Npath des classes modèles
COCM	Couplage moyen entre les objets des classes modèles
CMCM	Cohésion moyenne entre les méthodes des classes modèles
NCV	Nombre de classes vues
CCV	Complexité moyenne des classes vues
CNCV	Complexité moyenne de type Npath des classes vues
COCV	Couplage moyen entre les objets des classes vues
CMCV	Cohésion moyenne entre les méthodes des classes vues
NCC	Nombre de classes contrôleurs
CCC	Complexité moyenne des classes contrôleurs
CNCC	Complexité moyenne de type Npath des classes contrôleurs
COCC	Couplage moyen entre les objets des classes contrôleurs
CMCC	Cohésion moyenne entre les méthodes des classes contrôleurs
NCP	Nombre de classes présentateurs
CCP	Complexité moyenne des classes présentateurs
CNCP	Complexité moyenne de type Npath des classes présentateurs
COCP	Couplage moyen entre les objets des classes présentateurs
CMCP	Cohésion moyenne entre les méthodes des classes présentateurs
NCVM	Nombre de classes vue-modèles
CCVM	Complexité moyenne des classes vue-modèles
CNCVM	Complexité moyenne de type Npath des classes vue-modèles
COCVM	Couplage moyen entre les objets des classes vue-modèles
CMCVM	Cohésion moyenne entre les méthodes des classes vue-modèles

5.2.3 Objectif

Cette étape vise à prédire le(s) patron(s) appliqué(s) dans une application donnée à l'aide des trois classificateurs entraînés lors des étapes précédentes.

5.2.4 Méthodologie

Étant donné le code source d'une application et ses métriques OO, nous commençons la détection par l'identification des classes du modèle. Pour cela, toutes les classes de l'application sont injectées au classificateur de modèle. Ce dernier se charge de séparer les classes de modèle des classes non-modèles. Ensuite, en utilisant les classes du modèle, nous calculons les métriques du code source pour chaque classe non-modèle, et nous les injectons dans le classificateur des autres rôles afin de prédire leurs rôles. Enfin, nous rassemblons les résultats des classifications précédentes en un seul vecteur de caractéristiques, et nous l'injectons au classificateur de patrons. Ce dernier produit la prédiction finale de(s) patron(s) appliqué(s) dans l'application.

5.3 Validation et expérimentations

Dans cette section, nous décrivons le protocole expérimental que nous avons suivi pour évaluer l'efficacité de l'approche MlCoach dans l'identification de multiples patrons architecturaux dans les applications Android. Plus précisément, nos expérimentations visent à répondre aux questions de recherche suivantes :

1. **QR1 :** Comment la classification multi-label se comporte-t-elle dans la détection de combinaison de patrons de type MVC ?
2. **QR2 :** Comment plusieurs patrons de type MVC peuvent-ils coexister au sein d'une même application ? Cette question peut être affinée en :
   a. **QR2.1 :** Quels patrons de type MVC ont tendance à coexister dans les applications Android ?
   b. **QR2.2 :** Quels sont les facteurs les plus fréquents qui conduisent à l'apparition de multiples patrons de type MVC dans la même application ?

Pour répondre à la première question de recherche (QR1), nous commençons par l'entraînement de nos classificateurs sur un ensemble de données, appelé l'ensemble de données MlCoach et décrit dans la sous-section 5.3.1. Nous rapportons les performances de classification en termes de

métriques d'évaluation présentées dans la sous-section 5.3.3. Ensuite, nous évaluons l'approche MLCOACH sur l'ensemble de données de validation (décrit dans la section 5.3.2) et comparons les résultats obtenus avec RIMAZ (Daoudi *et al.*, 2019) et COACH. Nous présentons les résultats obtenus en termes d'exactitude, de précision, de rappel et de F1. Nos ensembles de données sont disponibles dans le paquet de réplication [16].

Pour répondre à la deuxième question de recherche (QR2), nous nous appuyons sur les résultats de la validation de la question (QR1), comme l'illustre la figure 5.5. En particulier, pour répondre à la question QR2.1, nous identifions et analysons les applications de l'ensemble de données de validation qui appliquent plusieurs patrons. Pour une application donnée, nous considérons qu'elle applique un patron spécifique si son score, donné par MLCOACH, dépasse un certain seuil. Pour définir le seuil pour chaque patron (c'est-à-dire MVC, MVP et MVVM), nous avons analysé toutes les applications de l'ensemble de données de validation pour lesquelles MLCOACH a correctement détecté le patron appliqué et choisi la valeur minimale des scores comme seuil pour ce patron. Pour le patron MVC, nous avons utilisé la valeur minimale obtenue à partir de la liste des applications qui utilisent MVC, ce qui correspond à $MVC_{threshold} = 0.213$. De même, nous avons calculé les seuils pour MVP et MVVM, respectivement, et obtenu les seuils suivants : $MVP_{threshold} = 0.206$ et $MVVM_{threshold} = 0.211$. Ainsi, une application est considérée comme mélangeant plusieurs patrons si au moins deux des scores rapportés par MLCOACH sont supérieurs à leurs seuils correspondants. En utilisant la valeur minimale pour chaque patron, nous avons étendu notre analyse à tous les cas possibles, même avec un faible pourcentage. L'objectif est d'obtenir une vue d'ensemble de la combinaison de plusieurs patrons.

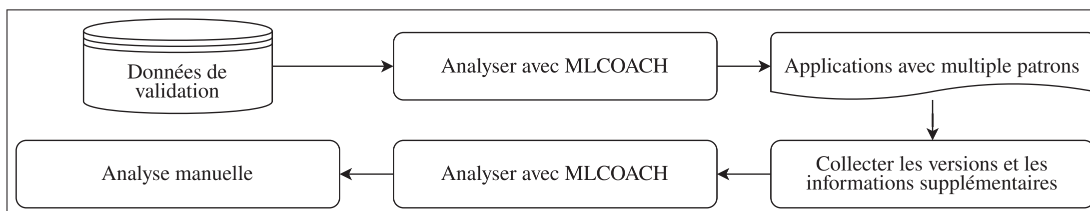

Figure 5.5 Processus de l'analyse qualitative

[16] https ://github.com/ChaimaChekhaba/MLCOACH

Pour obtenir plus d'informations sur la question QR2.1 et répondre à la question QR2.2, nous avons utilisé MlCoach pour analyser deux versions ultérieures pour chaque application qui applique plusieurs patrons. Ainsi, en plus de la version actuelle, nous avons collecté sur GitHub la version précédente de ces applications. Nous avons également récupéré certaines informations sur ces dernières et leurs versions sélectionnées qui sont pertinentes pour notre analyse. Il s'agit de la *date de publication*, de la *taille* (#loc) et du *nombre de contributeurs* (#contr). Plus précisément, la *date de publication* peut être utile pour comprendre les nouvelles tendances dans l'adoption des patrons architecturaux, car plusieurs composants et bibliothèques ont été introduits ces dernières années qui ciblent l'architecture des applications Android (par exemple, *JetPack*). La *taille* de l'application peut également avoir un lien avec la combinaison de plusieurs patrons. En effet, les petites applications ont tendance à ne pas appliquer de patrons du tout. Les grandes applications, en revanche, sont plus complexes et peuvent donc utiliser et mélanger des patrons architecturaux. L'expérience des *contributeurs* peut également avoir un impact sur la combinaison des patrons architecturaux. Ensuite, nous avons analysé manuellement les deux versions de chaque application en analysant le code source de chaque version, le fichier README.MD correspondant et les commentaires sur les modifications (commits). Nous avons concentré notre analyse sur les classes qui ont changé de rôle d'une version à l'autre. À travers l'analyse de deux versions ultérieures, l'objectif est de comprendre si les développeurs ont intentionnellement appliqué plusieurs patrons dans leurs applications ou s'ils l'ont fait par inadvertance pour d'autres raisons.

5.3.1 Ensemble de données MlCoach

Pour entraîner les classificateurs de notre approche, nous nous sommes appuyés sur l'ensemble de données que nous avons construit dans la première étude (l'approche Coach). Cet ensemble dispose de 69 applications Android annotées manuellement. Chaque application est étiquetée avec le patron dominant appliqué, et chaque classe avec son rôle selon notre analyse manuelle. Avec un tel jeu de données, il est inutile d'effectuer la classification multi-label puisque chaque instance (classe ou application) a un seul label (rôle ou patron). Cela signifie que chaque

label (rôle ou patron) dans cet ensemble de données pourrait combiner plusieurs labels selon notre définition des labels. À cette fin, nous avons ré-annoté toutes les instances de l'ensemble de données afin de raffiner l'annotation en assignant pour chaque classe ses rôles possibles (selon notre définition des rôles), et pour chaque application le(s) patron(s) appliqué(s). Comme mentionné dans la section 5.1, nous avons organisé le nouvel ensemble de données annoté en trois ensembles de données : ensemble de données des classes modèles (DCM), ensemble de données des autres rôles (DAR) et ensemble de données de patrons (DP). Tableaux 5.3, 5.4 et 5.5 montrent la distribution des classes modèles, des classes avec d'autres rôles et des patrons respectivement dans les ensembles de données DCM, DAR et DP selon notre annotation manuelle.

Tableau 5.3 Distribution des classes dans l'ensemble de données DCM

#classes	#modèle	#non_modèle
410	200	210

Tableau 5.4 Distribution des classes avec d'autres rôles dans l'ensemble de données DAR

#classes	#vue	#contrôleur	#présentateur	#vue-modèle	#aucun	#vue+contrôleur
1822	230	66	56	37	1060	373

Tableau 5.5 Distribution des patrons dans l'ensemble de données DP

#apps	#mvc	#mvp	#mvvm	#none	#mvc+mvp	#mvc+mvvm
69	25	10	8	23	2	1

Au cours de l'apprentissage, nous avons divisé aléatoirement l'ensemble de données en deux parties : entraînement et test (80% pour l'entraînement et 20% pour le test) en respectant l'équilibre des données entre l'entraînement et le test. Nous avons également effectué la

validation croisée en utilisant la fonction `Kfold` de Scikit-learn (avec 10 essais) afin d'assurer une meilleure généralisation des classificateurs entraînés.

5.3.2 Données de validation

Pour évaluer l'approche MlCoach, nous nous sommes appuyés sur l'ensemble de données de validation que nous avons utilisé pour valider l'approche Coach. Cet ensemble contient 265 applications collectées à partir de GitHub et étiquetées avec le patron appliqué dominant. De cette liste, nous avons sélectionné 191 applications pour lesquelles nous avons pu calculer leurs métriques OO en utilisant l'outil Metrics-Reloaded (Jet-Brains-Marketplace, 2021).

5.3.3 Métriques d'évaluation

La prédiction d'un classificateur multi-label pour une entité (qu'il s'agisse d'une classe ou d'une application) est un ensemble de labels. À titre d'exemple, considérons une classe dans laquelle le développeur a combiné deux rôles : Contrôleur et Vue. Le classificateur d'autres rôles est censé renvoyer les deux rôles de la classe. En fonction de son résultat, trois scénarios peuvent être distingués :

1. Le classificateur identifie correctement les deux rôles, son résultat est alors entièrement correct ;
2. Le classificateur n'identifie qu'un seul rôle (par exemple, Vue) et ne parvient pas à identifier l'autre rôle (Contrôleur), son résultat est alors partiellement correct ;
3. Le classificateur ne parvient pas à identifier les deux rôles, son résultat est alors entièrement faux.

Dans leur forme originale, aucune des métriques d'évaluation traditionnelles (telles que la précision et le rappel) ne capture de tel concept. Ainsi, nous avons utilisé les formules fournies par (Sorower, 2016) qui ciblent la classification multi-label pour calculer les métriques d'évaluation, à savoir l'exactitude, la précision, le rappel, F1 et la fonction de perte de Hamming. Nous avons également considéré les intervalles de confiance pour la précision et la fonction de perte de

Hamming (95 %). En outre, nous avons effectué une évaluation par label afin de reporter avec précision les résultats de la classification multi-label. Pour ce cas, nous avons utilisé les formules standard pour les métriques d'évaluation (c.-à-d. l'exactitude, la précision, le rappel et le F1), car l'évaluation par label est équivalente à une évaluation de classification binaire.

5.4 Résultats et discussions

Dans cette section, nous discutons les résultats obtenus et répondons à nos questions de recherche.

5.4.1 QR1 : Comment la classification multi-label se comporte-t-elle dans la détection de combinaison de patrons de type MVC ?

Dans cette section, nous commençons par présenter et discuter les performances de chaque classificateur séparément. Ensuite, nous présentons et analysons les résultats de la comparaison de MlCoach avec Coach et Rimaz (Daoudi *et al.*, 2019).

5.4.1.1 Performance de la classification des classes modèles

Tableau 5.6 présente les performances en termes d'exactitude, de précision, de rappel et de mesure F1 des différentes stratégies et architectures expérimentées dans la classification des classes modèles. La stratégie basée sur des heuristiques est la moins performante, avec une valeur F1 de 44,6%. En fait, l'emploi des heuristiques qui utilisent l'analyse statique et inspectent un ensemble de bibliothèques de persistance de données dans le code ne permet pas la détection correcte des classes modèle dans les applications qui n'utilisent pas ces dernières. De plus, il n'est pas possible de couvrir toutes les bibliothèques qui peuvent être utilisées pour implémenter des classes modèles en Android. Les stratégies basées sur l'apprentissage, quant à elles, ont atteint des performances élevées, avec une valeur F1 de 89% et 97% pour MLP et CNN, respectivement. D'après ces résultats, les stratégies basées sur l'apprentissage sont nettement plus performantes que la stratégie basée sur des heuristiques (amélioration de +48,4% pour F1). Cela signifie que l'exploitation du vocabulaire utilisé dans les classes modèles est plus pertinente pour leur détection. En fait, les classes modèles ont tendance à utiliser le même vocabulaire

qui est généralement dérivé du domaine de l'application. Ainsi, l'application d'une stratégie basée sur l'apprentissage permet d'apprendre le vocabulaire commun sans avoir à spécifier toutes les bibliothèques utilisées pour implémenter les classes modèles. À titre d'exemple, la stratégie basée sur des heuristiques a échoué dans l'identification des classes modèles qui gèrent les données extraites à partir de fichiers et les objets de type *SharedPreferences* du cadre d'applications Android, alors que les stratégies basées sur l'apprentissage ont réussi à identifier ces cas [17].

Nos résultats indiquent également que la stratégie basée sur l'apprentissage profond est plus performante que la stratégie MLP. En fait, une architecture ANN entièrement connectée peut apprendre plus efficacement le vocabulaire général utilisé dans l'implémentation des classes modèles. En ajoutant des couches de convolution, les performances ont atteint les valeurs les plus élevées, avec une amélioration de +8% pour F1 par rapport à la stratégie MLP. Cela s'explique par la capacité du CNN à se concentrer sur les caractéristiques locales, c'est-à-dire à apprendre de petites parties du vocabulaire des classes modèles qui aident à différencier les classes modèles des classes non-modèles.

Tableau 5.6 Performance globale de la classification des classes modèles

Stratégie	Exactitude	Précision	Rappel	Mesure F1
Heuristiques	28.7%	34.8%	61.8%	44.6%
MLP	89%	91%	87%	89%
CNN	**97%**	**96%**	**99%**	**97%**

5.4.1.2 Performance de la classification des autres rôles

Tableaux 5.7 et 5.8 présentent les résultats obtenus de la classification de rôles. Le classificateur a atteint des performances très élevées avec une valeur moyenne de mesure F1 de 83% et une perte de Hamming de 0,055, ce qui est une valeur très faible. Les intervalles de confiance montrent

[17] Nous fournissons les résultats détaillés de la classification des classes modèles dans le paquet de réplication

que le classificateur est sûr à 95% que sa précision est supérieure à 91% et que son erreur est inférieure à 0,08.

Par rapport à chaque rôle individuellement, le tableau 5.7 montre que les rôles Vue, Contrôleur, Vue-modèle et Aucun sont détectés avec des performances élevées avec des valeurs F1 de 87,2%, 88,7%, 99% et 92,6%, respectivement. Nous remarquons que les rôles Vue et Contrôleur sont détectés avec des performances élevées de l'ordre de 87% de F1. Ces résultats montrent la pertinence des métriques proposées dans leur détection. En particulier, les métriques (NGUIC, NDS) et (NLIE, NIE_XML) ciblent les rôles Vue et Contrôleur, respectivement, ce qui nous a permis de distinguer correctement ces deux rôles des autres. Nous pensons également que la performance (valeur F1 de 99%) pour le rôle Vue-modèle est principalement due à l'indicateur `extendsAndroidViewModel` que nous avons utilisé comme partie du vecteur de caractéristiques pour l'apprentissage. Cependant, étant donné que toutes les classes ayant ce rôle dans l'ensemble de données MlCoach étendent la classe *ViewModel* du cadre d'applications Android, cela peut réduire la capacité du classificateur à détecter les classes ayant ce rôle qui n'étendent pas la classe *ViewModel*. Pour les classes ayant le rôle Aucun, les performances élevées du classificateur peuvent s'expliquer par le nombre de classes ayant ce rôle dans l'ensemble de données MlCoach. En fait, ce nombre est le plus élevé par rapport aux autres rôles, ce qui permet au classificateur de mieux apprendre les classes Aucun. Pour le rôle Présentateur, le classificateur a obtenu les pires performances, soit environ 32% de précision, 94% de rappel et 47% de F1. Ces résultats se justifient par deux raisons : 1) Le petit nombre de classes Présentateur dans l'ensemble de données MlCoach ; et 2) le manque de métriques de code appropriées à utiliser pour caractériser ce rôle. En fait, nous nous appuyons sur les métriques OO pour la détection de ce rôle, qui sont insuffisantes et doivent être améliorées. Nous devons clairement définir de nouvelles métriques qui combinent plusieurs métriques de dépendances directes et indirectes entre les rôles Vue et Modèle.

Tableau 5.7 Performance de la classification des autres rôles : évaluation par label

Rôle	Exactitude	Précision	Rappel	Mesure F1
Vue	91.2%	88.9%	85.8%	87.2%
Contrôleur	93.3%	87.4%	90.1%	88.7%
Présentateur	94.3%	**32.8%**	94.9%	**47.9%**
Vue-modèle	99.9%	99%	99.1%	99%
Aucun	93.3%	97.1%	88.5%	92.6%

Tableau 5.8 Performance globale de la classification des autres rôles

HL	Exc	Pr	R	F1	95% Conf int (Exc)	95% Conf int (HL)
0.055	94.4%	81%	91.5%	83.08%	[0.914 ; 0.973]	[0,026 ; 0.084]

5.4.1.3 Performance de la classification de patrons

Tableaux 5.9 et 5.10 présentent les performances obtenues de la classification de patrons sur l'ensemble de données MLCOACH (la partie test). Le classificateur a obtenu des performances très élevées avec une valeur moyenne de mesure F1 de 80,8% et une perte de Hamming de 0,125, ce qui est très faible. Les intervalles de confiance indiquent que le classificateur de patrons est sûr à 95% que sa précision est supérieure à 70% et que son erreur est inférieure à 0,29.

Concernant chaque patron individuellement, le patron MVVM est correctement détecté avec une valeur F1 de 100%. Cela s'explique par les performances très élevées des classificateurs précédents en matière d'identification des rôles des classes. En particulier, cela est lié à l'efficacité du classificateur des autres rôles dans l'identification des classes avec les rôles Vue-modèle et Vue. En effet, le fait de fournir des données d'entrée correctes au classificateur de patron lui a permis de bien généraliser.

Pour le patron MVC, le classificateur a obtenu la performance la plus faible, avec une valeur F1 de 66,6%, par rapport aux autres patrons. Il est intéressant de noter que le patron MVP est identifié avec une valeur F1 de 83,3%. Pour les applications sans patron (Aucun), le classificateur a obtenu des performances élevées avec une valeur F1 de 73,3%.

Dans l'ensemble, ces résultats montrent que le classificateur de patrons a la capacité de distinguer efficacement les différents patrons dans les applications Android. Ils montrent également la pertinence des métriques de code sélectionnées pour la tâche de détection de multiples patrons architecturaux dans les applications Android.

Tableau 5.9 Performance de la classification de patrons : évaluation par label

Patron	Exactitude	Précision	Rappel	Mesure F1
MVC	71.4%	66.6%	66.6%	66.6%
MVP	92.8%	100%	75%	83.3%
MVVM	100%	100%	100%	100%
Aucun	85.7%	58.3%	100%	73.3%

Tableau 5.10 Performance globale de la classification de patrons

HL	Exc	Pr	R	F1	95% Conf int (Exc)	95% Conf int (HL)
0.125	87.4%	81.2%	85.4%	80.8%	[0,7 ; 1.0]	[0.048 ; 0.298]

5.4.1.4 Comparaison de MlCoach, Coach et Rimaz

Tableau 5.11 présente les performances obtenues par MlCoach, Coach et Rimaz sur l'ensemble de données de validation. En ce qui concerne la détection du patron dominant dans les applications Android, MlCoach surpasse les autres outils dans la détection de tous les patrons, avec une amélioration de +9,8% de F1 (voir la ligne Moyenne dans le tableau 5.11). Si nous considérons uniquement la détection de MVC, MVP et sans patron (Aucun), MlCoach surpasse significativement MlCoach et Rimaz avec une amélioration moyenne de +12.5% de F1.

Pour le patron MVVM, COACH et MLCOACH ont obtenu des performances très proches autour de 84% de F1, par rapport à COACH qui a eu les meilleures performances. La combinaison des métriques OO avec l'indicateur `extendsAndroidViewModel` conduit MLCOACH à ajouter automatiquement les métriques OO dans la spécification du rôle Vue-modèle, malgré le fait que cet indicateur soit le plus pertinent selon l'ensemble de données MLCOACH. COACH, en revanche, s'appuie exclusivement sur cet indicateur, ce qui lui permet d'être un peu plus performant, car toutes les applications avec le patron MVVM dans l'ensemble de données de validation utilisent la classe `AndroidViewModel` du cadre d'applications Android. Il est intéressant de noter que RIMAZ n'a pas pu détecter le patron MVVM du tout, malgré l'utilisation d'une heuristique axée sur l'utilisation de la liaison de données par l'intermédiaire de la bibliothèque de liaison de données *Data binding* [18].

Tableau 5.11 Comparaison de MLCOACH, COACH et RIMAZ

Patron	MLCOACH			COACH			RIMAZ		
	Pr	R	F1	Pr	R	F1	Pr	R	F1
MVC	57.8%	92.8%	**71.2%**	51.9%	96.4%	67.5%	58.1%	89.2%	70.4%
MVP	89.4%	66.6%	**76.4%**	76.9%	39%	51.9%	47%	33.3%	39%
MVVM	86.9%	80%	**83.3%**	92.8%	78%	**84.7%**	-	-	-
Aucun	93.5%	93.5%	**93.5%**	76%	87%	81.2%	76.5%	33.3%	46.4%
Moyenne	81.9%	82.2%	**81.1%**	74.4%	75.1%	71.3%	45.4%	38.9%	38.9%

Ces résultats montrent l'efficacité des ANN pour de telles tâches, ainsi que le choix approprié des métriques de code prises en compte dans la détection. En effet, l'exploitation de différents types d'informations du code source, c'est-à-dire l'information sémantique et structurelle, s'est avérée très utile pour maximiser la spécification des différents rôles dans les patrons de type MVC et, par conséquent, pour détecter efficacement les patrons appliqués.

Quant à la détection du mélange de plusieurs patrons, MLCOACH était le seul outil capable de détecter efficacement l'application de plusieurs patrons dans la même application. Nous sommes

[18] https ://developer.android.com/topic/libraries/data-binding

conscients que les deux outils (Coach et Rimaz) ont été conçus pour détecter le patron dominant dans les applications Android, et il n'est donc pas surprenant qu'ils n'aient pas pu détecter ces cas. Cependant, fournir de telles informations aux développeurs, c'est-à-dire les patrons combinés dans une application, est plus précis et les aide à agir de manière appropriée.

Pour la tâche de détection des patrons de type MVC, l'outil MlCoach surpasse les outils existants avec une amélioration globale de +9,8% de F1. En outre, il permet de détecter le mélange de plusieurs patrons dans les applications Android. L'application de la classification multi-label a été bénéfique pour la détection du patron dominant ainsi que des autres patrons co-apparaissant en même temps dans les applications Android.

5.4.2 QR2 : Comment plusieurs patrons de type MVC peuvent-ils coexister au sein d'une même application ?

Nous répondons à la question QR2 en identifiant les patrons architecturaux qui ont tendance à co-apparaître dans une même application, et en analysant les facteurs qui conduisent à leur co-application dans une application.

5.4.2.1 QR2.1 : Quels patrons de type MVC ont tendance à coexister dans les applications Android ?

Figure 5.6 représente la distribution des patrons architecturaux parmi les applications étudiées selon MlCoach. Les résultats montrent que 24% des applications mélangent deux patrons différents. Il est intéressant de noter que le patron MVC apparaît dans toutes les applications, et qu'environ 63% d'entre elles combinent MVC et MVP, tandis que 37% combinent MVC et MVVM.

Pour en savoir plus, nous avons analysé les patrons appliqués dans les deux versions des applications qui appliquent plusieurs patrons. Ces applications sont répertoriées dans le tableau 5.12. Figures 5.7a et 5.7b montrent les patrons utilisés dans les versions antérieures et actuelles

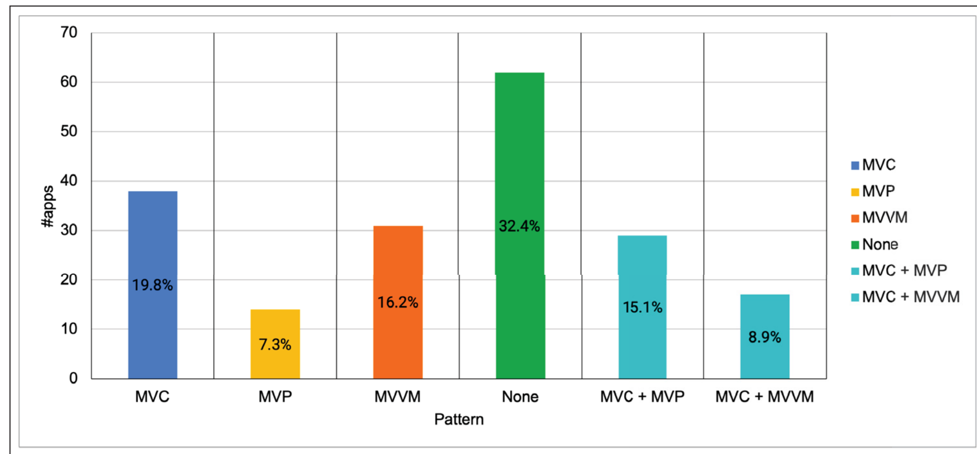

Figure 5.6 Distribution des patrons de type MVC parmi les applications étudiées selon les résultats de MLCOACH

de chaque application, respectivement. Ces figures indiquent les scores de similarité renvoyés par le classificateur pour chaque patron.

Nous avons remarqué que le patron MVC était toujours présent dans le $V_{antrieure}$ (avec des scores de similarité variés d'une application à l'autre) contrairement au MVP et au MVVM, qui étaient soit absents puis introduits dans le $V_{actuelle}$ (applications #6 à #8), soit présents puis supprimés du $V_{actuelle}$ (application #4), soit ont fait l'objet de changements (applications #1 à #3 et #5). De plus, pour trois des applications analysées (applications #6 à #8), les développeurs semblent introduire progressivement le patron MVP tout en continuant à utiliser le patron MVC. Ces résultats sont conformes à ceux rapportés dans des travaux antérieurs concernant la large adoption du patron MVC dans les applications Android (par exemple Daoudi *et al.* (2019); Campos *et al.* (2015)). Enfin, pour quasiment toutes les applications (à l'exception de l'application #4), MVC et MVP sont combinés dans leur version actuelle. Ce point sera approfondi dans notre analyse qualitative dans la sous-section suivante.

En résumé, MVC est présent dans toutes les applications Android qui mélangent plusieurs patrons architecturaux. En outre, il a tendance à persister même lorsque les développeurs adoptent MVP ou MVVM.

Tableau 5.12 Les résultats de l'analyse pour QR2

#	app	#cont	#versions	Date de publication	#loc	Patrons appliqués (scores de similarité)	Pourquoi multiples patrons ?
1	ShiZhong	2	V_current	2019-05-05	15076	82% (MVC), 78% (MVP)	Décision architecturale
			V_previous	2018-08-06	14396	67% (MVC), 98% (MVP)	
2	ThirtyInch	7	V_current	2020-01-28	8568	32% (MVC), 66% (MVP)	
			V_previous	2018-07-31	9396	32% (MVC), 70% (MVP)	
3	AppChooser	4	V_current	2020-10-14	3525	64% (MVC), 99% (MVP)	
			V_previous	2019-12-09	2871	58% (MVC), 93% (MVP)	
4	Graceful Movies	2	V_current	2019-02-27	5465	61% (MVC), 75% (MVVM)	Évolution de l'architecture de l'application
			V_previous	2019-01-12	5448	27% (MVC), 93% (MVP) 71% (MVVM)	
5	Android permission checker	4	V_current	2019-06-07	1932	34% (MVC), 93% (MVP)	
			V_previous	2017-10-06	1696	30% (MVC), 94% (MVP)	
6	United4	2	V_current	2018-10-23	5255	84% (MVC), 34% (MVP)	Mauvaise implémentation des patrons de type MVC
			V_previous	2018-01-16	3836	88% (MVC)	
7	Privacy friendly pedometer	7	V_current	2017-07-02	12036	87% (MVC), 24% (MVP)	
			V_previous	2016-11-05	7729	87% (MVC)	
8	Smart refresh layout	6	V_current	2020-03-18	27588	97% (MVC), 80% (MVP)	
			V_previous	2019-08-13	25938	99% (MVC)	

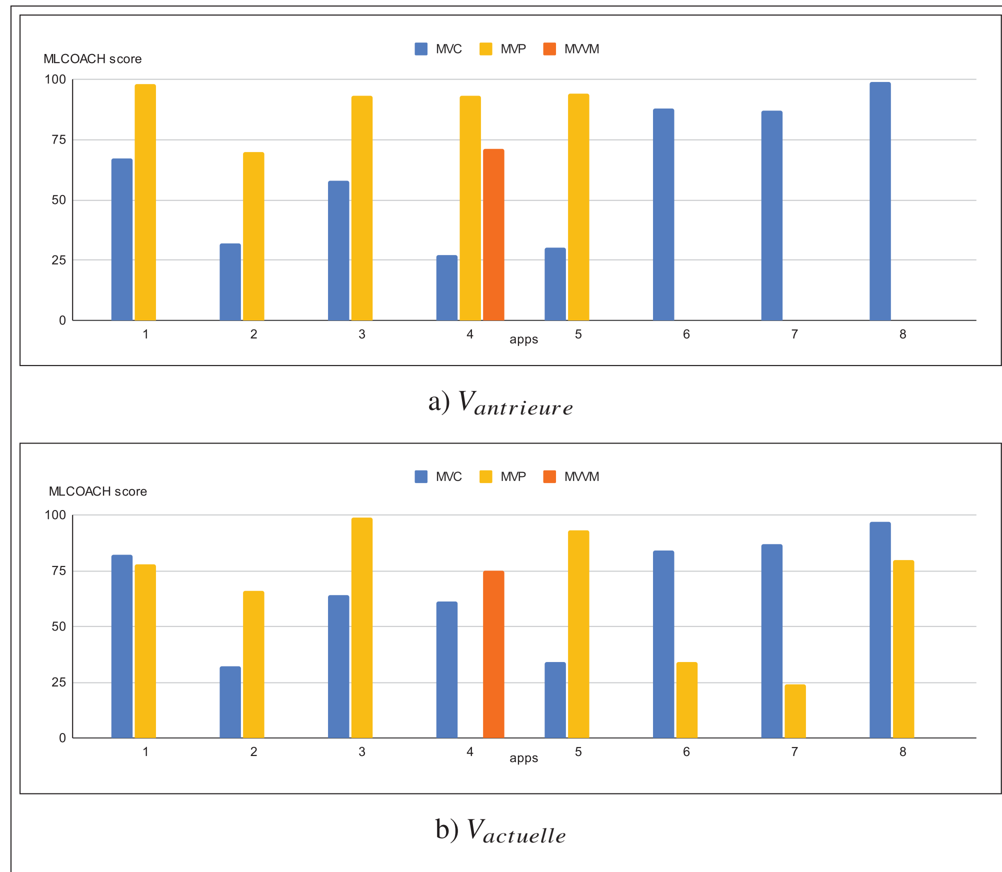

Figure 5.7 Les patrons appliqués dans les versions $V_{antrieure}$ et $V_{actuelle}$ des applications étudiées

5.4.2.2 QR2.2 : Quels sont les facteurs les plus fréquents qui conduisent à l'apparition de multiples patrons de type MVC dans la même application ?

Tableau 5.12 présente les informations collectées sur les applications qui mélangent plusieurs patrons, ainsi que les résultats de notre analyse manuelle (voir la colonne "Pourquoi plusieurs patrons ?"). Nous avons remarqué que les applications qui mélangent plusieurs patrons varient en taille (#loc) et en nombre de contributeurs (#cont). Toutes les applications analysées, y compris leurs versions, ont été publiées entre 2016 et 2020. Aucune corrélation n'a été observée entre les informations collectées et le mélange de patrons architecturaux dans ces applications.

À travers l'analyse manuelle, nous avons constaté que les deux patrons combinés dans les applications #1, #2 et #3, c'est-à-dire MVC et MVP, emploient des classes modèles différentes.

En fait, la partie MVC de l'application #1 utilise des classes modèles implémentées en tant que classes de données récupérées via une API (par exemple, la classe *InputNameModel*). Par contre, la partie MVP utilise d'autres classes modèles définies comme de simples classes modèles (classes *Bean*) et persistantes dans une base de données locale (par exemple, la classe *MonthSalaryEntity*). Comme pour les applications #2 et #3, les patrons MVC et MVP sont implémentés dans deux modules séparés (qui sont *sample* et *library* dans l'application #2 et *app* et *library* dans l'application #3), chacun avec ses propres classes modèles. Le patron MVC, implémenté dans *sample* et *app*, est utilisé pour gérer la logique métier la plus simple (par exemple, lire et afficher les données des classes modèles), tandis que le patron MVP, utilisé dans *library*, implémente une logique métier plus complexe. En outre, nous n'avons observé aucun changement dans les rôles des classes dans les deux versions subséquentes. En fait, toutes les modifications apportées à ces versions visaient à corriger des bogues d'affichage ou à étendre les fonctionnalités de base. Sur la base de ces résultats, nous avons conclu que les patrons MVC et MVP ont été combinés avec intention dans les applications #1, #2 et #3, et nous reconnaissons cela comme une décision architecturale dans le tableau 5.12.

Pour les applications #4 et #5, nous avons remarqué que les classes qui font partie de l'instance MVC sont implémentées en tant qu'activités et implémentent des fonctionnalités simples (par exemple, l'affichage de données statiques). En fait, les fonctionnalités principales de ces applications, où les données sont manipulées et où la plupart des interactions avec l'utilisateur sont gérées, ont été implémentées dans les classes qui font partie des instances MVP/MVVM. Nous avons également observé que la plupart des modifications apportées au code au fil des différentes versions sont effectuées dans ces classes. En termes de changements de rôles, nous n'avons observé aucun changement dans les rôles de toutes les classes des deux applications. En fait, les classes MVC/MVP/MVVM étaient présentes dans les anciennes versions, bien qu'avec peu de fonctionnalités implémentées. En outre, les changements reportés sont destinés à compléter les fonctionnalités de ces classes. C'est pourquoi nous avons conclu que ces deux applications étaient en cours de migration vers MVVM (application #4) et MVP (application

#5). Nous avons validé cette conclusion à partir des commentaires des contributeurs, car ils ont spécifiquement indiqué leur objectif de migrer vers le patron MVP/MVVM.

Quant aux applications #6, #7 et #8, nous avons constaté que la combinaison de plusieurs patrons était implémentée dans le même module. En outre, la manière dont les classes d'interaction avec l'utilisateur accèdent aux classes modèles est différente d'une classe à l'autre. En fait, certaines classes d'interaction avec l'utilisateur utilisent les classes présentateurs pour accéder aux classes modèles. D'autres classes d'interaction avec l'utilisateur accèdent directement aux classes modèles en faisant référence à leurs méthodes. Par exemple, dans l'application 6, la classe d'interaction avec l'utilisateur *UscriptActivity* accède aux classes modèles par l'intermédiaire de la classe présentateur *ThreadWatcher*. Parallèlement, la classe d'interaction avec l'utilisateur *SettingListFragment* manipule la classe modèle *Setting* par le biais de ses méthodes. Un tel comportement indique une mauvaise implémentation du patron MVP, puisque les présentateurs sont déclarés, mais ne sont pas utilisés de manière appropriée. Nous avons validé ces conclusions, car les développeurs ont indiqué dans leurs commentaires qu'ils appliquaient le patron MVP. Cela peut s'expliquer par le fait que le patron MVP est plus difficile à implémenter que le patron MVC, ce qui conduit les développeurs à prendre des raccourcis plus simples. Selon les contributeurs de l'application #2 [19], le patron MVP est plus difficile à implémenter par rapport au MVC et au MVVM après l'introduction de la bibliothèque *JetPack*. La gestion du cycle de vie des présentateurs est une tâche difficile. En particulier, lorsque l'écran est tourné, le système recrée une nouvelle activité qui doit restaurer les données de l'activité détruite.

En ce qui concerne les classes qui ont changé de rôle au cours des différentes versions des applications 6, 7 et 8, la plupart d'entre elles étaient de simples vues qui sont devenues des contrôleurs, car elles accédaient directement aux classes modèles. Nous avons également trouvé des classes qui n'ont pas de rôle spécifique et qui sont devenues des présentateurs, puisqu'elles ont été introduites en tant que classes présentateurs. Cependant, l'ensemble des changements visait à corriger des bogues et ne mettait pas l'accent sur l'architecture.

---

[19] L'application #2 a une note de 1 K étoiles sur GITHUB

Nous avons aussi observé des faits intéressants en ce qui concerne la conception de base de ces applications. Les trois applications souffrent d'une mauvaise séparation des préoccupations. En fait, certains de leurs activités et leurs fragments ne respectent pas le principe de la responsabilité unique, avec des centaines de #loc qui pourraient être divisés en fonctionnalités plus simples à l'aide de fragments et de vues du cadre d'applications Android. En outre, l'application #8 déclare certaines classes modèles et classes d'interaction avec l'utilisateur dans les mêmes fichiers, comme dans la classe *BannerPracticeActivity* qui déclare les classes modèles *Movie* et *BannerItem* dans le même fichier également, ce qui constitue une très mauvaise organisation de l'application.

Pour résumer nos conclusions, de multiples patrons architecturaux peuvent apparaître dans une application en raison de :

1. **Une décision architecturale prise par les développeurs ;**
2. **Une mise à niveau de l'architecture de l'application pour appliquer un patron architectural spécifique ;**
3. **Une mauvaise implémentation des patrons de type MVC.**

5.5 Conclusion

Dans ce chapitre, nous avons présenté notre approche MLCOACH qui vise à identifier la combinaison de patrons architecturaux de type MVC dans les applications Android en utilisant la classification multi-label. Nous avons ensuite présenté en détail les différents classificateurs construits, les données d'apprentissage ainsi que les métriques d'évaluation. Après, nous avons évalué MLCOACH sur un ensemble de 191 applications Android et l'avons comparé avec l'approche COACH et l'outil RIMAZ. Les résultats obtenus sont prometteurs et surpassent les deux autres approches. Également, nous avons mené une analyse qualitative visant à comprendre les facteurs favorisant l'émergence des implémentations combinant de multiples patrons de type MVC. Nos résultats suggèrent que toutes ces applications combinent le patron MVC avec une de ses variantes. De plus, ce mélange peut être le résultat d'une décision architecturale prise par

les développeurs, la mise à niveau de l'architecture de l'application pour appliquer un patron architectural spécifique ou une mauvaise implémentation des patrons de type MVC.

CHAPITRE 6

MENACES À LA VALIDITÉ

Dans cette section, nous discutons des menaces qui pourraient influencer la validité des résultats de notre étude.

6.1 Validité de construction

Les menaces à la validité de construction concernent la relation entre la théorie et l'observation. Cela pourrait être lié aux métriques de code utilisées pour la classification. Nous avons utilisé toutes les métriques standards qui peuvent être calculées avec l'outil Metrics-Reloaded, un plugiciel libre bien établi. Nous avons également utilisé les outils Paprika et Spoon (Pawlak, Monperrus, Petitprez, Noguera & Seinturier, 2016) pour calculer les métriques proposées, deux outils bien reconnus dans la communauté de recherche en génie logiciel, ce qui atténue les éventuels problèmes de fiabilité. De plus, les métriques et les caractéristiques utilisées dans notre approche ont été choisies en fonction de leur utilisation persistante dans la littérature. Nous devons cependant souligner qu'en considérant que les classes Vue-modèle s'étendent à partir du cadre d'applications Android, nous limitons notre étude à la détection des instances de MVVM dans lesquels les développeurs utilisent la classe `AndroidViewModel` du cadre. Cependant, comme toutes les applications MVVM de l'ensemble de données d'apprentissage et de l'ensemble de données de validation utilisent la classe `AndroidViewModel` du cadre, nous avons choisi de nous concentrer sur l'implémentation largement utilisée du patron MVVM dans les applications Android. Cependant, une étude complémentaire pourrait être lancée pour approfondir ce point.

6.2 Validité interne

Les menaces à la validité interne font référence à l'existence d'une relation causale entre l'expérience et les résultats obtenus. Dans notre travail, cela pourrait être lié aux ensembles de données adoptés. Comme mentionné précédemment, les deux ensembles de données utilisés pour

l'apprentissage (Coach et MlCoach) ont été construits manuellement par deux des chercheurs. L'annotation manuelle effectuée par chaque chercheur était basée sur les mêmes définitions et les concepts. Pourtant, les résultats pourraient être biaisés en raison des différentes implémentations de chaque patron architectural. Pour atténuer cette menace, chaque chercheur a validé les annotations de son collègue et nous avons également envoyé une enquête aux développeurs pour nous assurer que l'architecture que nous avons identifiée correspond aux patrons architecturaux dominants prévus par le développeur de l'application. Nous mettons également à disposition les deux ensembles de données d'apprentissage Coach et MlCoach ainsi que la justification de l'annotation dans le paquet de réplication [20].

Une autre menace est liée aux biais qui auraient pu être introduits dans la sélection des ensembles de test et d'entraînement. Cette menace a été traitée en randomisant l'échantillonnage de l'ensemble de données. Et comme la taille de l'échantillon est relativement petite, nous avons utilisé la validation croisée `k-fold` pour s'assurer que chaque enregistrement de l'ensemble de données est affecté de manière aléatoire à la fois à l'étape d'entraînement et de détection.

La fiabilité des résultats de l'analyse qualitative constitue également une autre menace de notre travail. Une étude complémentaire devrait être lancée pour approfondir ce point, notamment à travers l'analyse de nos conclusions avec des praticiens.

6.3 Validité externe

Les menaces à la validité externe font référence à la capacité de généraliser les résultats de notre approche. Cela pourrait faire référence au déséquilibre de l'ensemble de données d'apprentissage et au nombre insuffisant d'instances MVP et MVVM. Comme mentionné avant, MVC est le patron architectural le plus utilisé dans Android. En outre, les patrons MVP et MVVM ont été introduits récemment, et les développeurs les trouvent relativement plus difficiles à apprendre et à appliquer, surtout avec la documentation limitée sur Internet. Pour atténuer cette menace, nous avons utilisé la technique d'*échantillonnage de la classe minoritaire* (*Up-sample Minority*

[20] Paquet de réplication est disponible dans https://github.com/ChaimaChekhaba/COACH

Class) pour équilibrer l'ensemble de données d'apprentissage. Cette menace peut également être liée au fait que nous avons construit l'ensemble de données avec des applications libres provenant de différentes communautés Android. Toutefois, nos résultats peuvent être généralisés aux applications à code source fermé et aux projets industriels étant donné qu'ils utilisent le même cadre de développement et les mêmes bibliothèques qui sont principalement à code source ouvert.

CONCLUSION ET RECOMMANDATIONS

La notion d'architecture d'applications mobiles a toujours suscité l'intérêt des développeurs et des chercheurs depuis l'avènement des applications mobiles. De fait, la nature des applications mobiles, ainsi que le contexte de développement, nécessitent une attention particulière à la conception des applications mobiles, notamment à l'application des patrons architecturaux de type MVC, les patrons les plus adoptés dans le développement mobile.

À travers notre revue de littérature, nous avons montré l'importance d'analyser les patrons de type MVC dans les applications Android, la nécessité de proposer des approches de détection automatique des différentes implémentations et potentiellement l'identification de mélange de multiples patrons de type MVC dans les applications Android. Nous avons également soulevé la problématique de limitation des approches de détection existantes dans la littérature et montré la nécessité de proposer une approche de détection automatique indépendamment de l'implémentation des patrons architecturaux.

Dans ce travail, nous avons proposé deux approches outillées : Coach et MlCoach. L'approche Coach permet la détection de patron du type MVC dominant dans les applications Android à l'aide de la classification traditionnelle. Quant à l'approche MlCoach, elle permet la détection de mélange de plusieurs patrons de type MVC en utilisant la classification multi-label basée sur les réseaux de neurones artificiels et l'apprentissage profond. Nous avons également proposé de nouvelles métriques de code source spécifiques à Android visant à mesurer l'interaction avec l'utilisateur, la logique de présentation et la manipulation de la logique métier.

En l'absence de données de référence standards, nous avons effectué nos analyses sur un ensemble de données annotées manuellement contenant 69 applications Android libres écrites en Java. Les deux approches ont été évaluées sur 191 applications Android et les résultats obtenus ont été comparés à ceux de Rimaz, le seul outil existant dans la littérature. Nos résultats suggèrent

que nos deux approches sont plus performantes que l'outil Rimaz. De plus, MlCoach est en mesure d'identifier le mélange de plusieurs patrons au sein de la même application.

Plus encore, nous avons effectué une analyse manuelle de huit applications Android ainsi que de leur historique mélangeant plusieurs patrons de type MVC au sein de la même application. Nous avons trouvé que toutes ces applications combinent MVC avec une de ses variantes. De plus, le mélange est le résultat d'une décision architecturale prise par les développeurs, la mise à niveau de l'architecture de l'application pour appliquer un patron architectural spécifique ou d'une mauvaise implémentation des patrons de type MVC.

En synthèse, notre recherche aide les développeurs mobiles à comprendre et à faire évoluer correctement leurs applications d'un point de vue architectural. Elle permet également aux chercheurs d'établir une caractérisation objective de l'état de l'art et des pratiques actuelles en matière de conception d'applications mobiles.

Dans le cadre de travaux futurs, nous envisageons en premier lieu d'analyser nos résultats avec des praticiens, notamment pour valider l'outil MlCoach et les résultats de l'analyse manuelle. Nous compléterons aussi notre analyse manuelle par une étude empirique de l'impact de mélange des patrons de type MVC sur la qualité des applications mobiles. À moyen terme, nous prévoyons d'élargir l'ensemble de données d'apprentissage en ajoutant davantage d'applications Android provenant de différents dépôts d'applications et même d'inclure des applications à code source fermé. À long terme, étant donné que Kotlin est devenu le langage de programmation officiel pour le développement Android, nous nous efforcerons à analyser les applications Android écrites en Kotlin et à reproduire notre étude dessus. Nous planifions aussi d'étudier à grande échelle comment les patrons de type MVC sont implémentés par la communauté et d'identifier les violations les plus courantes dans leurs implémentations.

ANNEXE I

CLASSIFICATION MULTI-LABEL

L'apprentissage supervisé traditionnel est l'un des paradigmes d'apprentissage automatique les plus étudiés. Il s'agit de modéliser chaque objet du monde réel par une seule instance en lui associant un label unique (Zhang & Zhou, 2014). En d'autres termes, dans l'apprentissage supervisé traditionnel, une instance est considérée comme appartenant à une seule signification sémantique. Cependant, les objets du monde réel peuvent être plus complexes et avoir plusieurs significations sémantiques simultanément. C'est de là que le concept d'apprentissage multi-label est né. Historiquement, les premières applications de l'apprentissage multi-label concernaient la tâche de catégorisation de documents texte (Tsoumakas & Katakis, 2007), car les documents texte appartiennent généralement à plusieurs classes conceptuelles. Ces dernières années, l'apprentissage supervisé multi-label a attiré l'attention de la communauté de l'apprentissage automatique et a été largement appliqué à divers problèmes tels que l'annotation automatique de contenu multimédia (Boutell, Luo, Shen & Brown, 2004; Qi *et al.*, 2007; Sanden & Zhang, 2011), la bio-informatique (Clare & King, 2001; Elisseeff & Weston, 2001; Zhang & Zhou, 2006) et l'exploration de contenu Web (Kazawa, Izumitani, Taira & Maeda, 2004; Tang, Rajan & Narayanan, 2009). Dans ce travail, nous nous intéressons à une tâche particulière de l'apprentissage supervisé multi-label : la classification multi-label. Cette dernière est une variante de la classification traditionnelle, qui permet d'attribuer à chaque instance plusieurs labels non exclusifs. Les applications modernes la requièrent de plus en plus, notamment dans la classification de caractéristiques de protéines (Zhang & Zhou, 2005), la catégorisation de la musique (Li & Ogihara, 2003) et la classification sémantique de scènes (Boutell *et al.*, 2004).

Dans la suite du chapitre, nous présentons d'une manière formelle les concepts de base de la classification multi-label et les métriques d'évaluation.

1. Définitions

Étant donné un ensemble d'exemples d'apprentissage D composé de paires (x_i, y_i) tel que x_i un vecteur de caractéristiques et $y_i \in L$ le label qui lui est associé, avec L un ensemble de labels disjoints et $|L| > 1$.

La tâche de classification consiste à trouver une fonction $f(x)$ qui associe chaque vecteur de caractéristiques x_i à sa classe associée y_i, avec $i = 1, 2, ..., n$ où $n = |D|$ est le nombre total d'exemples d'apprentissage.

$$f(x_i) = y_i, \;\; avec \;\; y_i \in L \tag{A I-1}$$

- Si $|y_i| = 1 \;\; \forall (x_i, y_i) \in D$, alors on parle de classification uni-label ou traditionnelle,
- Si $\exists (x_i, y_i) \in D \;\; tel \; que \;\; |y_i| > 1$, alors on parle de classification multi-label.

De manière générale, les problèmes de classification peuvent être catégorisés selon le nombre de labels dans l'ensemble L :

- Si $|L| = 2$ alors, il s'agit d'un problème de classification binaire. Les deux labels dans ce cas ont des sémantiques opposées,
- Si $|L| > 2$ alors, il s'agit d'un problème de classification multi-classe. Généralement, les labels sont mutuellement exclusifs.

La classification multi-label est une généralisation de la classification multi-classe dans laquelle les labels ne sont pas exclusifs (de Carvalho & Freitas, 2009). Cela signifie qu'un tel problème peut être traité en tant que multiples problèmes de classification binaire séparés, où chaque problème est défini pour un seul label. Il s'agit de la stratégie la plus populaire dans la résolution des problèmes de la classification multi-label (Tsoumakas & Katakis, 2007).

2. Métriques d'évaluation

Dans la classification multi-label, la prédiction d'une instance est un ensemble de labels ; par conséquent, la prédiction peut être totalement correcte, partiellement correcte (avec différents

niveaux de correction) ou totalement fausse (Sorower, 2016). Dans leur forme originale, aucune des métriques d'évaluation traditionnelles (telles que la précision et le rappel) ne capture de tel concept. Ainsi, pour évaluer la classification multi-label, nous nous sommes appuyés sur les métriques d'évaluation présentées dans le travail de Sorower (2016).

2.1 Évaluation globale

Soit D un ensemble de données d'évaluation multi-label avec n instances $(x_i, \ ; Y_i)$, $1 \leq i \leq n, \ ; x_i \in X, \ ; Y_i \in Y = 0, 1^k$ avec k = nombre de labels. Soit h un classificateur multi-label et $Z_i = h(x_i) = 0, 1^k$ l'ensemble des appartenances aux labels prédits par h pour l'exemple x_i.

- **Perte de Hamming (ou Hamming Loss) :** rapporte combien de fois, en moyenne, la pertinence d'un exemple pour un label de classe est prédite de manière incorrecte (Sorower, 2016). Par conséquent, la perte de Hamming prend en compte l'erreur de prédiction (un label incorrect est prédit) et l'erreur manquante (un label pertinent n'est pas prédit), normalisées par rapport au nombre total de classes et au nombre total d'exemples.

$$HL = \frac{1}{kn} \sum_{i=1}^{n} \sum_{l=1}^{k} [I(l \in Z_i \wedge l \notin Y_i) + I(l \notin Z_i \wedge l \notin Y_i) \tag{A I-2}$$

  Plus la valeur de la perte de Hamming est petite, meilleures sont les performances de la classification.

- **Exactitude (ou Accuracy) :** définie comme la proportion des labels correctement prédits par rapport au nombre total (prédit et réel) de labels pour l'instance.

$$A = \frac{1}{n} \sum_{i=1}^{n} \frac{|Y_i \cap Z_i|}{|Y_i \cup Z_i|} \tag{A I-3}$$

- **Précision (P) :** définie par la proportion des labels corrects prédits par rapport au nombre total de labels réels, en moyenne sur toutes les instances.

$$P = \frac{1}{n} \sum_{i=1}^{n} \frac{|Y_i \cap Z_i|}{|Z_i|} \tag{A I-4}$$

- **Rappel (R) :** est la proportion de labels correctement prédits par rapport au nombre total de labels prédits, en moyenne sur toutes les instances.

$$R = \frac{1}{n} \sum_{i=1}^{n} \frac{|Y_i \cap Z_i|}{|Y_i|} \tag{A I-5}$$

- **Mesure F1 :** la moyenne harmonique de la précision et du rappel.

$$F1 = 2 * \frac{P * R}{P + R} = \frac{1}{n} \sum_{i=1}^{n} \frac{2 * |Y_i \cap Z_i|}{|Y_i| + |Z_i|} \tag{A I-6}$$

2.2 Évaluation par label

Dans ce cas, les métriques d'évaluation sont les métriques de classification standards, qui sont :

- **Exactitude (A, l) :** définie comme la proportion de la prédiction correcte de l par rapport au nombre total de labels (prédits et réels) pour l'instance.

$$A, l = \frac{1}{n} \sum_{i=1}^{n} |Y_i[l] \cap Z_i[l]| \tag{A I-7}$$

- **Précision (P, l) :** définie comme la proportion de la prédiction correcte de l par rapport au nombre total de l réels, en moyenne sur toutes les instances.

$$P, l = \frac{1}{n} \sum_{i=1}^{n} \frac{|Y_i[l] \cap Z_i[l]|}{|Z_i[l]|} \tag{A I-8}$$

- **Rappel (R, l) :** est la proportion de la prédiction correcte de l par rapport au nombre total de prédictions de l, en moyenne sur toutes les instances.

$$R, l = \frac{1}{n} \sum_{i=1}^{n} \frac{|Y_i[l] \cap Z_i[l]|}{|Y_i[l]|} \tag{A I-9}$$

- **Mesure F1 (F1, l) :** la moyenne harmonique de la précision et du rappel.

$$F1, l = 2 * \frac{(P, l) * (R, l)}{(P, l) + (R, l)} \tag{A I-10}$$

BIBLIOGRAPHIE

42matters. (2020a, April, 13). Google Play Store ASO with App Update Frequency Statistics 2021 [Website]. Repéré à https://42matters.com/google-play-aso-with-app-update-frequency-statistics.

42matters. (2020b, April, 13). Apple App Store ASO with App Update Frequency Statistics 2021 [Website]. Repéré à https://42matters.com/apple-app-store-aso-with-app-update-frequency-statistics.

Alencar, P. S., Cowan, D. D. & Lucena, C. J. P. d. (1996). A formal approach to architectural design patterns. *International Symposium of Formal Methods Europe*, pp. 576–594.

Alexander, C. (1977). *A pattern language : towns, buildings, construction*. Oxford university press.

Alon, U., Brody, S., Levy, O. & Yahav, E. (2018a). code2seq : Generating sequences from structured representations of code. *arXiv preprint arXiv :1808.01400*.

Alon, U., Zilberstein, M., Levy, O. & Yahav, E. (2018b). code2vec : Learning Distributed Representations of Code.

Alon, U., Zilberstein, M., Levy, O. & Yahav, E. (2019). code2vec : Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL), 1–29.

Amorim, L., Costa, E., Antunes, N., Fonseca, B. & Ribeiro, M. (2015). Experience report : Evaluating the effectiveness of decision trees for detecting code smells. *2015 IEEE 26th international symposium on software reliability engineering (ISSRE)*, pp. 261–269.

and practices, M. . P. (2010, January, 14). Microsoft Application Architecture Guide, 2nd Edition [Microsoft pattern and practices website]. Repéré à https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ee658117(v=pandp.10).

Apple, d. (2021, May, 17). Displaying and Managing Views with a View Controller [Apple developer website]. Repéré à https://developer.apple.com/documentation/uikit/view_controllers/displaying_and_managing_views_with_a_view_controller.

Aritra, R. (2019). Ultimate Android Reference. Repéré à https://github.com/aritraroy/UltimateAndroidReference.

Aritra, R. (2018, Feb, 01). Ultimate Android Reference. [Github repository]. Repéré à https://github.com/aritraroy/UltimateAndroidReference/blob/master/README.md.

Avgeriou, P. & Zdun, U. (2005). Architectural patterns revisited-a pattern language.

Azadi, U., Fontana, F. A. & Zanoni, M. (2018). Poster : machine learning based code smell detection through WekaNose. *2018 IEEE/ACM 40th International Conference on Software Engineering : Companion Proceedings (ICSE-Companion)*, pp. 288–289.

Azadi, U., Fontana, F. A. & Taibi, D. (2019). Architectural smells detected by tools : a catalogue proposal. *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*, pp. 88–97.

Bagheri, H., Garcia, J., Sadeghi, A., Malek, S. & Medvidovic, N. (2016). Software architectural principles in contemporary mobile software : from conception to practice. *Journal of Systems and Software*, 119, 31–44.

Barbez, A., Khomh, F. & Guéhéneuc, Y.-G. (2019). Deep Learning Anti-patterns from Code Metrics History. *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 114–124.

Boutell, M. R., Luo, J., Shen, X. & Brown, C. M. (2004). Learning multi-label scene classification. *Pattern recognition*, 37(9), 1757–1771.

Bower, A. & McGlashan, B. (2000). Twisting the triad. *Tutorial Paper for European Smalltalk User Group (ESUP)*.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M. et al. (2008). *Pattern-Oriented Software Architecture : A System of Patterns, Volume 1*. John wiley & sons.

Cai, Y. & Kazman, R. (2017). Detecting and quantifying architectural debt : theory and practice. *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pp. 503–504.

Campos, E., Kulesza, U., Coelho, R., Bonifácio, R. & Mariano, L. (2015). Unveiling the Architecture and Design of Android Applications. *Proceedings of the 17th International Conference on Enterprise Information Systems*, 2, 201–211.

Campos., E., Kulesza., U., Coelho., R., Bonifácio., R. & Mariano., L. (2015). Unveiling the Architecture and Design of Android Applications - An Exploratory Study. *Proceedings of the 17th International Conference on Enterprise Information Systems - Volume 1 : ICEIS,*, pp. 201-211. doi : 10.5220/0005398902010211.

Carvalho, S. G., Aniche, M., Veríssimo, J., Durelli, R. S. & Gerosa, M. A. (2019). An empirical catalog of code smells for the presentation layer of Android apps. *Empirical Software Engineering*, 24(6), 3546–3586.

Chang, C.-H., Lu, C.-W., Lin, C.-H., Yang, M.-F. & Tsai, C.-F. (2008). An Experience of Applying Pattern-based Software Framework to Improve the Quality of Software Development : 4. The Design and Implementation of OS2F. *Journal of Software Engineering Studies, Vol. 2, No. 6*.

Chekhaba, C., Rebatchi, H., ElBoussaidi, G., Moha, N. & Kpodjedo, S. (2021). Coach : classification-based architectural patterns detection in Android apps. *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, pp. 1429–1438.

Chen, W., Zhang, M. & Zhang, Y. (2014). Distributed feature representations for dependency parsing. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 23(3), 451–460.

Chihada, A., Jalili, S., Hasheminejad, S. M. H. & Zangooei, M. H. (2015). Source code and design conformance, design pattern detection from source code by classification approach. *Applied Soft Computing*, 26, 357–367.

Clare, A. & King, R. D. (2001). Knowledge discovery in multi-label phenotype data. *European conference on principles of data mining and knowledge discovery*, pp. 42–53.

Clark, M. & Perez, P.-L. (2022, March, 17). JDepend tool [website]. Repéré à http://www.testingtoolsguide.net/tools/jdepend/.

Cruz, L. & Abreu, R. (2018). Using automatic refactoring to improve energy efficiency of android apps. *arXiv preprint arXiv :1803.05889*.

Cruz, L. & Abreu, R. (2019). Catalog of energy patterns for mobile applications. *Empirical Software Engineering*, 24(4), 2209–2235.

Dagger. (2021, May, 9). Dagger library [Github repository]. Repéré à https://google.github.io/dagger/.

Daoudi, A., ElBoussaidi, G., Moha, N. & Kpodjedo, S. (2019). An exploratory study of MVC-based architectural patterns in Android apps. *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, pp. 1711–1720.

Databinding. (2016). Android Data Binding Library [Documentation officielle]. Repéré à https://developer.android.com/topic/libraries/data-binding.

de Carvalho, A. C. & Freitas, A. A. (2009). A tutorial on multi-label classification techniques. *Foundations of computational intelligence volume 5*, 177–195.

Dey, T. (2011). A Comparative Analysis on Modeling and Implementing with MVC Architecture. *IJCA Proceedings on International Conference on Web Services Computing (ICWSC)*, 1, 44–49.

Documentation-Android. (2021a). The Activity lifecycle. Repéré à https://developer.android.com/guide/components/activities/activity-lifecycle.

Documentation-Android. (2021b). Input events overview. Repéré à https://developer.android.com/guide/topics/ui/ui-events.

Documentation-Android. (2022, April, 13). Platform Architecture [Documentation officielle]. Repéré à https://developer.android.com/guide/platform.

Documentation-Microsoft, w. (2022, September, 22). Desktop Guide (WPF .NET) [Website]. Repéré à https://learn.microsoft.com/en-us/dotnet/desktop/wpf/overview/?view=netdesktop-6.0.

Dwivedi, A. K., Tirkey, A. & Rath, S. K. (2019). Applying learning-based methods for recognizing design patterns. *Innovations in Systems and Software Engineering*, 15(2), 87–100.

Elisseeff, A. & Weston, J. (2001). A kernel method for multi-labelled classification. *Advances in neural information processing systems*, 14.

Ernst, N. A., Bellomo, S., Ozkaya, I., Nord, R. L. & Gorton, I. (2015a). Measure It ? Manage It ? Ignore It ? Software Practitioners and Technical Debt. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, (ESEC/FSE 2015), 50–60. doi : 10.1145/2786805.2786848.

Ernst, N. A., Bellomo, S., Ozkaya, I., Nord, R. L. & Gorton, I. (2015b). Measure it ? manage it ? ignore it ? software practitioners and technical debt. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pp. 50–60.

Evansdata. (2021, March, 17). The Evansdata Website [website]. Repéré à https://evansdata.com/press/viewRelease.php?pressID=244.

F-droid. (2021, May, 20). F-droid website [Website]. Repéré à https://f-droid.org/en/about/.

Fontana, F. A., Mäntylä, M. V., Zanoni, M. & Marino, A. (2016). Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, 21(3), 1143–1191.

Fontana, F. A., Lenarduzzi, V., Roveda, R. & Taibi, D. (2019). Are architectural smells independent from code smells ? An empirical study. *Journal of Systems and Software*, 154, 139–156.

Fowler, M. (1997). Refactoring : Improving the design of existing code. *11th European Conference. Jyväskylä, Finland*.

Fowler, M. (2018). *Refactoring : improving the design of existing code*. Addison-Wesley Professional.

Fu, S. & Shen, B. (2015). Code bad smell detection through evolutionary data mining. *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 1–9.

Gamma, E. (1995). *Design patterns : elements of reusable object-oriented software*. Pearson Education India.

Ganesh, S., Sharma, T. & Suryanarayana, G. (2013). Towards a Principle-based Classification of Structural Design Smells. *J. Object Technol.*, 12(2), 1–1.

Gao, J., Li, L., Bissyandé, T. F. & Klein, J. (2019). On the Evolution of Mobile App Complexity. *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pp. 200-209. doi : 10.1109/ICECCS.2019.00029.

Garcia, J., Popescu, D., Edwards, G. & Medvidovic, N. (2009a). Toward a Catalogue of Architectural Bad Smells. *Architectures for Adaptive Software Systems*, pp. 146–162.

Garcia, J., Popescu, D., Edwards, G. & Medvidovic, N. (2009b). Identifying Architectural Bad Smells. *2009 13th European Conference on Software Maintenance and Reengineering*, pp. 255-258. doi : 10.1109/CSMR.2009.59.

Garlan, D., Allen, R. & Ockerbloom, J. (1994). Exploiting style in architectural design environments. *ACM SIGSOFT software engineering notes*, 19(5), 175–188.

Guamán, D., Delgado, S. & Pérez, J. (2021). Classifying model-view-controller software applications using self-organizing maps. *IEEE Access*, 9, 45201–45229.

HARUN, F. B. (2019). Review of Ios Architectural Pattern For Testability, Modifiability, And Performance Quality. *Journal of Theoretical and Applied Information Technology*, 97(15).

Heaton-Research. (2021, April, 21). The Number of Hidden Layers. Repéré à https://www.heatonresearch.com/2017/06/01/hidden-layers.html.

Hecht, G., Benomar, O., Rouvoy, R., Moha, N. & Duchien, L. (2015). Tracking the software quality of android applications along their evolution (t). *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 236–247.

Hecht, G., Moha, N. & Rouvoy, R. (2016). An empirical study of the performance impacts of android code smells. *Proceedings of the international conference on mobile software engineering and systems*, pp. 59–69.

Imran, A. (2019). Design Smell Detection and Analysis for Open Source Java Software. *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 644-648. doi : 10.1109/ICSME.2019.00104.

Jet-Brains-Marketplace. (2021, April, 8). MetricsReloaded plugin. Repéré à https://plugins.jetbrains.com/plugin/93-metricsreloaded.

Jetpack. (2017, April, 13). Android Jetpack [Documentation officielle]. Repéré à https://developer.android.com/jetpack.

Kaczor, O., Guéhéneuc, Y.-G. & Hamel, S. (2010). Identification of design motifs with pattern matching algorithms. *Information and Software Technology*, 52(2), 152–168.

Katakis, I., Tsoumakas, G. & Vlahavas, I. (2008). Multilabel text classification for automated tag suggestion. *Proceedings of the ECML/PKDD*, 18, 5.

Kazawa, H., Izumitani, T., Taira, H. & Maeda, E. (2004). Maximal margin labeling for multi-topic text categorization. *Advances in neural information processing systems*, 17.

Kessentini, M. & Ouni, A. (2017). Detecting Android Smells Using Multi-Objective Genetic Programming. *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pp. 122-132. doi : 10.1109/MOBILESoft.2017.29.

Khomh, F., Vaucher, S., Guéhéneuc, Y.-G. & Sahraoui, H. (2011). BDTEX : A GQM-based Bayesian approach for the detection of antipatterns. *Journal of Systems and Software*, 84(4), 559–572.

Koenig, A. (1995). Patterns and antipatterns. *Journal of Object-Oriented Programming*, 8(1), 46–48.

Komolov, S., Dlamini, G., Megha, S. & Mazzara, M. (2022). Towards Predicting Architectural Design Patterns : A Machine Learning Approach. *Computers*, 11(10), 151.

Krasner, G. E. (1988). A cookbook for using model-view-controller user interface paradigmin smalltalk-80. *J. Object Oriented Programming*, 1(3), 26–49.

Kreimer, J. (2005). Adaptive detection of design flaws. *Electronic Notes in Theoretical Computer Science*, 141(4), 117–136.

Kuhn, M. & Johnson, K. (2013). Remedies for Severe Class Imbalance. Dans *Applied Predictive Modeling* (pp. 419–443). New York, NY : Springer New York.

LaMarche, J. & Mark, D. (2010). More iPhone 3 Development : Tackling IPhone SDK 3.

Le, D. M., Carrillo, C., Capilla, R. & Medvidovic, N. (2016). Relating architectural decay and sustainability of software systems. *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pp. 178–181.

Le Metayer, D. (1998). Describing software architecture styles using graph grammars. *IEEE Transactions on Software Engineering*, 24(7), 521-533. doi : 10.1109/32.708567.

Li, T. & Ogihara, M. (2003). Detecting emotion in music.

Linares-Vásquez, M., Klock, S., McMillan, C., Sabané, A., Poshyvanyk, D. & Guéhéneuc, Y.-G. (2014). Domain matters : bringing further evidence of the relationships among anti-patterns, application domains, and quality-related metrics in java mobile apps. *Proceedings of the 22nd International Conference on Program Comprehension*, pp. 232–243.

Lippert, M. & Roock, S. (2006). *Refactoring in large software projects : performing complex restructurings successfully*. John Wiley & Sons.

Liu, H., Xu, Z. & Zou, Y. (2018). Deep learning based feature envy detection. *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pp. 385–396.

Logarix. (2021, May, 20). AI Reviewer [website]. Repéré à https://www.aireviewer.com/doc/analyses/.

Macia, I., Arcoverde, R., Garcia, A., Chavez, C. & Von Staa, A. (2012). On the relevance of code anomalies for identifying architecture degradation symptoms. *2012 16Th european conference on software maintenance and reengineering*, pp. 277–286.

Maiga, A., Ali, N., Bhattacharya, N., Sabané, A., Guéhéneuc, Y.-G. & Aimeur, E. (2012). Smurf : A svm-based incremental anti-pattern detection approach. *2012 19th Working Conference on Reverse Engineering*, pp. 466–475.

Manning, C., Raghavan, P. & Schütze, H. (2010). Introduction to information retrieval. *Natural Language Engineering*, 16(1), 100–103.

Mark, R. (2015). Software Architecture Patterns-Understanding Common Architecture Patterns and When to Use Them. O'Reilly, Febrero.

Martinez, P. (2021, July, 9). Hexagonal Architecture, there are always two sides to every story [Website]. Repéré à https://medium.com/ssense-tech/hexagonal-architecture-there-are-always-two-sides-to-every-story-bc0780ed7d9c.

Mendoza, A. (2013). *Mobile user experience : patterns to make sense of it all*. Newnes.

Microsoft-Silverlight, w. (2022, September, 22). Microsoft Silverlight [Website]. Repéré à https://www.microsoft.com/silverlight/.

Mikolov, T., Chen, K., Corrado, G. & Dean, J. (2013). Efficient estimation of word representations in vector space. *arXiv preprint arXiv :1301.3781*.

Moha, N., Guéhéneuc, Y.-G., Duchien, L. & Le Meur, A.-F. (2009). Decor : A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1), 20–36.

Mosby. (2018, Dec, 17). A Model-View-Presenter and Model-View-Intent library for Android apps [Github repository]. Repéré à https://github.com/sockeqwe/mosby.

Moxy. (2021, May, 9). Moxy library [Github repository]. Repéré à https://github.com/moxy-community/Moxy.

MSDN, M. (2019, April, 10). Introduction to Model/View/ViewModel pattern for building WPF apps [Microsoft's MSDN]. Repéré à https://docs.microsoft.com/en-us/archive/blogs/johngossman/introduction-to-modelviewviewmodel-pattern-for-building-wpf-apps.

Nord, R. L., Ozkaya, I., Kruchten, P. & Gonzalez-Rojas, M. (2012). In search of a metric for managing architectural technical debt. *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, pp. 91–100.

Palomba, F., Di Nucci, D., Panichella, A., Zaidman, A. & De Lucia, A. (2017). Lightweight detection of android-specific code smells : The adoctor project. *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*, pp. 487–491.

Pawlak, R., Monperrus, M., Petitprez, N., Noguera, C. & Seinturier, L. (2016). Spoon : A library for implementing analyses and transformations of java source code. *Software : Practice and Experience*, 46(9), 1155–1179.

Pixelmatters, w. (2020, August, 7). VIPER design pattern [Website]. Repéré à https://www.pixelmatters.com/blog/tutorial-part-1-viper-design-pattern-what-when-why-and-how.

Potel, M. (1996). MVP : Model-View-Presenter the Taligent programming model for C++ and Java. *Taligent Inc*, 20.

Qi, G.-J., Hua, X.-S., Rui, Y., Tang, J., Mei, T. & Zhang, H.-J. (2007). Correlative multi-label video annotation. *Proceedings of the 15th ACM international conference on Multimedia*, pp. 17–26.

Reenskaug, T. (1979). THING-MODEL-VIEW-EDITOR-an Example from a planningsystem. Technical note, Xerox PARC.(May 1979).

Reimann, J., Brylski, M. & Aßmann, U. (2014). A tool-supported quality smell catalogue for android developers. *Proc. of the conference Modellierung 2014 in the Workshop Modellbasierte und modellgetriebene Softwaremodernisierung–MMSM*, 2014.

Rimawi, D. & Zein, S. (2019). A Model Based Approach for Android Design Patterns Detection. *2019 3rd International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT)*, pp. 1-10. doi : 10.1109/ISMSIT.2019.8932921.

Robert, C. M. (2012, August, 13). The Clean Code Blog [Website]. Repéré à https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html.

Roveda, R., Fontana, F. A., Pigazzini, I. & Zanoni, M. (2018). Towards an architectural debt index. *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pp. 408–416.

RxJava. (2021, Oct, 12). RxJava : Reactive Extensions for the JVM [Github repository]. Repéré à https://github.com/ReactiveX/RxJava.

Sanden, C. & Zhang, J. Z. (2011). Enhancing multi-label music genre classification through ensemble techniques. *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*, pp. 705–714.

Schmidt, D. C., Stal, M., Rohnert, H. & Buschmann, F. (2013). *Pattern-oriented software architecture, patterns for concurrent and networked objects*. John Wiley & Sons.

Schmitt Laser, M., Medvidovic, N., Le, D. M. & Garcia, J. (2020). ARCADE : an extensible workbench for architecture recovery, change, and decay evaluation. *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1546–1550.

Sharma, T. (2016). Designite - A Software Design Quality Assessment Tool. Repéré à https://doi.org/10.5281/zenodo.2566832.

Sharma, T., Mishra, P. & Tiwari, R. (2016). Designite : A software design quality assessment tool. *Proceedings of the 1st International Workshop on Bringing Architectural Design Thinking into Developers' Daily Activities*, pp. 1–4.

Shaw, M. & Garlan, D. (1994). An introduction to software architecture.

Simon, F., Seng, O. & Mohaupt, T. (2006). *Code-Quality-Management : technische Qualität industrieller Softwaresysteme transparent und vergleichbar gemacht*. dpunkt-Verlag.

Sokolova, K., Lemercier, M. & Garcia, L. (2013). Android passive MVC : a novel architecture model for the android application development. *International Conference on Pervasive Patterns and Applications*, pp. 7–12.

Sokolova, K., Lemercier, M., Garcia, L. & Saint Luc, L. C. (2014). Towards high quality mobile applications : Android passive MVC architecture. *International Journal On Advances in Software*, 7(2), 123–138.

SonarQube. (2022, April, 10). Code Quality and Code Security [Site web]. Repéré à https://www.sonarqube.org/.

Soot. (2022, April, 10). Soot - A framework for analyzing and transforming Java and Android applications [Site web]. Repéré à https://soot-oss.github.io/soot/.

Sorower, M. (2016). A Literature Survey on Algorithms for Multi-label Learning.

Stackoverflow. (2010, May, 27). MVC pattern on Android [Website]. Repéré à https://stackoverflow.com/questions/2925054/mvc-pattern-on-android.

Statista. (2020, April, 13). Global mobile OS market share in sales to end users from 1st quarter 2009 to 2nd quarter 2018 [Website]. Repéré à https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/.

Statista. (2021, April, 19). Frequency of operating system updates among smartphone owners in the United States, as of 2016 [Website]. Repéré à https://www.statista.com/statistics/747607/united-states-survey-smartphone-users-update-frequency-operating-system/.

Statista. (2022a). Mobile operating systems' market share worldwide from January 2012 to January 2022 [Website]. Repéré à https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/.

Statista. (2022b). Android - Statistics Facts [Website]. Repéré à https://www.statista.com/topics/876/android/.

Tang, L., Rajan, S. & Narayanan, V. K. (2009). Large scale multi-label classification via metalabeler. *Proceedings of the 18th international conference on World wide web*, pp. 211–220.

Taylor, R. N., Medvidovic, N. & Dashofy, E. M. (2009). Software architecture : foundations, theory, and practice.(2009). *Google Scholar Google Scholar Digital Library Digital Library*.

Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H. & Noble, J. (2010). The Qualitas Corpus : A curated collection of Java code for empirical studies. *2010 Asia Pacific Software Engineering Conference*, pp. 336–345.

Tsantalis, N., Chatzigeorgiou, A., Stephanides, G. & Halkidis, S. T. (2006). Design pattern detection using similarity scoring. *IEEE transactions on software engineering*, 32(11), 896–909.

Tsoumakas, G. & Katakis, I. (2007). Multi-label classification : An overview. *International Journal of Data Warehousing and Mining (IJDWM)*, 3(3), 1–13.

Venkatesh, G., Sampson, J., Goulding-Hotta, N., Venkata, S. K., Taylor, M. B. & Swanson, S. (2011). QsCores : Trading dark silicon for scalable energy efficiency with quasi-specific cores. *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 163–174.

Verdecchia, R., Malavolta, I., Lago, P. et al. (2019). Guidelines for architecting android apps : A mixed-method empirical study. *2019 IEEE International Conference on Software Architecture (ICSA)*, pp. 141–150.

Vladislav, B. (2019). Android Arsenal. Repéré à https://android-arsenal.com/.

Vladislav, B. (2018, Feb, 01). Android Arsenal. [Website]. Repéré à https://android-arsenal.com/.

Wang, C. (2020, Mars, 28). MVC, MVP, and MVVM on Android [Website]. Repéré à https://medium.com/\spacefactor\@m{}chauyan/mvc-mvp-and-mvvm-on-android-ac88e0cd6891.

Wasserman, A. I. (2010). Software engineering issues for mobile application development. *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pp. 397–400.

Wells, D. & Williams, L. (2003). *Extreme Programming and Agile Methods-XP/Agile Universe 2002 : Second XP Universe and First Agile Universe Conference Chicago, IL, USA, August 4-7, 2002. Proceedings*. Springer.

Wong, S., Cai, Y., Kim, M. & Dalton, M. (2011). Detecting software modularity violations. *Proceedings of the 33rd International Conference on Software Engineering*, pp. 411–420.

Yu, L.-C., Wang, J., Lai, K. R. & Zhang, X. (2017). Refining word embeddings using intensity scores for sentiment analysis. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 26(3), 671–681.

Zanoni, M., Fontana, F. A. & Stella, F. (2015). On applying machine learning techniques for design pattern detection. *Journal of Systems and Software*, 103, 102–117.

Zhang, M.-L. & Zhou, Z.-H. (2005). A k-nearest neighbor based algorithm for multi-label classification. *2005 IEEE international conference on granular computing*, 2, 718–721.

Zhang, M.-L. & Zhou, Z.-H. (2006). Multilabel neural networks with applications to functional genomics and text categorization. *IEEE transactions on Knowledge and Data Engineering*, 18(10), 1338–1351.

Zhang, M.-L. & Zhou, Z.-H. (2014). A Review on Multi-Label Learning Algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 26(8), 1819-1837. doi : 10.1109/TKDE.2013.39.

Zhou, G., Xie, Z., He, T., Zhao, J. & Hu, X. T. (2016). Learning the multilingual translation representations for question retrieval in community question answering via non-negative matrix factorization. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 24(7), 1305–1314.