# Gestion des contributions architecturales dans les projets logiciels : Métriques, analyses empiriques et apprentissage machine

Quentin Perez

THÈSE POUR OBTENIR LE GRADE DE DOCTEUR
DE L'INSTITUT MINES-TÉLÉCOM (IMT) -
ÉCOLE NATIONALE SUPÉRIEURE DES MINES D'ALÈS (IMT MINES ALÈS)

En Informatique

École doctorale : I2S – Information, Structures, Systèmes
Portée par l'Université de Montpellier

Unité de recherche : EuroMov Digital Health In Motion

# Gestion des contributions architecturales dans les projets logiciels : Métriques, analyses empiriques et apprentissage machine


Présentée par Quentin PEREZ

Le 13 décembre 2021

Sous la direction de Christelle URTADO
et Sylvain VAUTTIER


Devant le jury composé de :

| | |
|---|---|
| Xavier BLANC, PR, LABRI, Université de Bordeaux | Rapporteur |
| Anne ETIEN, PR, CRIStAL, Université de Lille | Rapportrice |
| Antoine BEUGNARD, PR, Lab-STICC, IMT Atlantique | Examinateur, Président |
| Chouki TIBERMACINE, MCF HDR, LIRMM, Université de Montpellier | Examinateur |
| Christelle URTADO, MA HDR, EuroMov DHM, IMT Mines Alès | Directrice |
| Sylvain VAUTTIER, MA HDR, EuroMov DHM, IMT Mines Alès | Directeur |


*“Je suis de ceux qui pensent que la science est d’une grande beauté.*
*Un scientifique dans son laboratoire est non seulement un technicien :*
*il est aussi un enfant placé devant des phénomènes naturels qui*
*l’impressionnent comme des contes de fées.”*

Marie CURIE

# *Remerciements*

Je tiens tout d'abord à remercier les directeurs de l'unité, Stéphane Perrey et Jacky Montmain pour m'avoir permis de réaliser ma thèse au sein d'EuroMov Digital Health in Motion. Jacky, ton soutien, ta confiance et les discussions que nous avons partagées m'ont fait grandir durant ces trois ans. Ainsi, tu as contribué à la réussite de cette thèse et à mon épanouissement dans l'unité.

Je tiens à remercier l'ensemble des membres du jury d'avoir accepté d'évaluer mon travail. Malgré le contexte sanitaire difficile, ils ont fait en sorte que je soutienne dans les meilleures conditions. Je remercie les Professeurs Anne Etien et Xavier Blanc d'avoir rapporté mon travail. Je remercie également le Professeur Antoine Beugnard, président du jury, et le Maître de Conférences HDR Chouki Tibermacine d'avoir accepté d'examiner mon travail.

Je souhaite tout particulièrement remercier mes directeurs de thèse Christelle Urtado et Sylvain Vauttier sans qui je n'aurais pu vivre cette aventure. Ils ont toujours su me faire confiance tout au long de mes trois ans de thèse et leur soutien m'a été d'une aide précieuse. Leurs qualités scientifiques, pédagogiques et humaines ont été déterminantes pour mon travail. Ainsi, je peux dire aujourd'hui que je n'avais pas seulement deux très bons directeurs de thèse mais aussi deux très bons collègues avec qui j'ai eu énormément plaisir à travailler. Les moments amicaux que nous avons passés autour d'un café à échanger ont contribué à ma motivation et à mon avancée.

Je remercie l'ensemble des personnels techniques et administratifs de l'unité et en particulier Edith Teychené pour son soutien dans toutes les épreuves administratives auxquelles j'ai pu être confronté et également sa bonne humeur en toutes circonstances. Je remercie aussi Pierre Jean et Pierre-Antoine Jean pour leurs conseils et leur support pour résoudre les questions techniques que j'ai pu rencontrer, ainsi que pour leur sympathie.

Je remercie l'ensemble des enseignants-chercheurs et enseignants du CERIS de m'avoir fait confiance pour dispenser TD et TP. Merci également pour les différents conseils qu'ils m'ont dispensés. J'en profite aussi pour remercier l'ensemble des étudiants que j'ai eu en TP et TD.

Je n'oublie pas non plus les collègues de bureau et doctorants avec qui j'ai partagé de chaleureux moments. Alexandre et Pierre-Antoine, vous avez droit à une dédicace spéciale ! Je n'oublierai jamais les parties de nerf dans les couloirs, les cafés (parfois bruyants mais drôles) ou encore les karaokés sur *Mauvaise foi nocturne* durant le temps de midi-deux. Merci également à tous les autres et en particulier à Cécile, Lucie, Camille, Valentin, Léa, Yann, Clara, Emilie, Thibault, Clément, Roland, Pascale et toutes celles et ceux que j'ai pu côtoyer tout au long de ma thèse.

Enfin je souhaite remercier grandement ma famille pour son soutien indéfectible durant cette thèse. Pap's, Belle-maman, Mél', Juju, Titou, Gweninou, Mich' et Mo', je vous dédie cette thèse car, sans vous et votre amour, elle ne serait pas ce qu'elle est. Vous avez contribué à ma réussite et je vous en serai éternellement reconnaissant. Merci à Marie de m'avoir supporté (dans tous les sens du terme) au quotidien durant cette thèse.

— ❤ sur vous tous —

# *Résumé*

En génie logiciel, l'outillage des processus de développement est aujourd'hui indispensable à la gestion qualitative des projets et peut couvrir différentes phases : conception, codage ou maintenance. Cet outillage protéiforme regroupe les mesures appelées métriques, les systèmes automatisés de production de code et les systèmes d'aide à la décision, prédictifs ou non. Le champ d'application de ces derniers est large : détection de *bad smells*, classification de tickets, etc. Dans cette thèse, nous nous intéressons à la production de métriques portant sur la conduite des projets logiciels en s'appuyant sur les contributions au code source et plus particulièrement à l'architecture *runtime* des applications. Nous utilisons également des méta-données liées à l'historique du projet et à son développement.

La première problématique étudiée concerne la classification de tickets logiciels décrivant des bogues. Nous entraînons un réseau de neurones sur un jeu de données reconnu par la communauté scientifique et nous optimisons ses hyper-paramètres à l'aide d'un algorithme génétique. De cette manière, nous obtenons des performances de classification supérieures à l'état de l'art. Ces performances permettent d'envisager l'intégration de notre classifieur dans des outils de gestion de projets ou dans un assistant à la rédaction de tickets.

La deuxième question abordée est le *turn-over* des développeurs contribuant au développement de l'architecture logicielle. La détection des développeurs contribuant de manière majeure ou mineure a déjà été formulée et des métriques dédiées proposées. Cependant, les approches existantes ne se focalisent pas spécifiquement sur le développement de l'architecture. C'est pourquoi nous avons créé un modèle formel ainsi qu'une métrique de contribution au code de l'architecture. Nous validons cette métrique de manière empirique puis nous proposons une analyse des différentes catégories de développeurs extraites par notre métrique. Les résultats mettent en lumière la présence de catégories de développeurs spécifiques dont un noyau de développeurs expérimentés contribuant de manière majeure au code de l'architecture durant toute la vie du projet.

La détection des développeurs expérimentés ayant de potentielles connaissances en architectures est également un problème que nous étudions. Contrairement aux approches existantes, nous utilisons l'apprentissage supervisé pour apprendre des profils de développeurs expérimentés. Aucun jeu de données étiquetées n'étant disponible dans la littérature scientifique, nous avons créé un jeu de données dédié. Pour cela nous extrayons plusieurs centaines de développeurs issus de 17 projets open source. Nous calculons ensuite 23 métriques pour chacun des développeurs contribuant à ces projets. Enfin, nous étiquetons notre jeu de données manuellement en utilisant les réseaux sociaux professionnels ainsi que la documentation des projets. Nous entraînons ensuite un classifieur sur ce jeu de données et nous l'évaluons. Nos résultats montrent de bonnes performances sur la détection de profils de développeur expérimentés, ce qui permet d'envisager l'utilisation d'un tel outil pour faciliter le management des projets.

Enfin, nous réalisons une étude par croisement des résultats obtenus via la métrique de contribution à l'architecture et ceux obtenus par la classification automatique des développeurs. Cette étude permet d'analyser le profil des développeurs expérimentés. Nous analysons également l'évolution des 23 métriques des développeurs durant leur passage d'inexpérimenté à expérimenté. Les résultats montrent qu'une grande part des développeurs expérimentés sont également des contributeurs majeurs à l'architecture et que le changement de catégorie d'un développeur est multifactoriel.

Nous fournissons également une évaluation de la reproductibilité de nos travaux en utilisant des cadres méthodologiques définis pour les études en génie logiciel empirique et en apprentissage automatique.

# Table des matières

# Table des figures

# Liste des tableaux

# Chapitre 1

# Introduction

Ce chapitre présente le contexte général de cette thèse. Il donne également une vision globale des problèmes traités, des contributions proposées et détaille le plan de ce manuscrit.

## 1.1 Contexte général

En informatique, les systèmes de gestion de versions et les dépôts de code logiciel sont les entrepôts virtuels de la production logicielle. Ces systèmes peuvent être locaux, sur un poste de développement, ou en ligne, permettant alors la collaboration des développeurs au sein des projets. Les systèmes de gestion de versions comme SVN ou CVS ont émergé dans les années 90. Ces derniers permettent de gérer un historique du code par sauvegarde successive des modifications. Au début des années 2000, le système de gestion de versions Git fait son apparition. Ce système est rapidement adopté par les développeurs dans les années qui suivent et contribue à l'émergence de plate-formes collaboratives telles que que Bitbucket, GitLab ou GitHub. Ces plate-formes permettent de distribuer le code source d'un logiciel mais également d'y contribuer. Elles entraînent la démocratisation du code en sources ouvertes (*open-source*). Ces plate-formes gèrent également tout un ensemble de données gravitant autour de la production du code qui sont appelées méta-données. Ces méta-données couvrent l'historique des modifications du projet, la liste des développeurs ou encore les tickets logiciels. Les tickets sont des données textuelles permettant aux utilisateurs des plate-formes collaboratives d'effectuer des demandes diverses : amélioration / ajout d'une fonctionnalité, description d'un bogue, mise à jour d'un élément du code, etc. De ce fait, les dépôts logiciels sont des sources de données permettant de suivre à la fois les évolutions du code source d'un projet mais aussi les évolutions du projet lui-même au travers de ses méta-informations.
Ces sources de données que sont les dépôts de code ont permis la création d'une nouvelle discipline, le génie logiciel empirique. Le génie logiciel s'intéresse à la création de méthodes et outils permettant la maîtrise de la qualité et des coûts de développement des logiciels. En génie logiciel empirique, une connaissance est construite par observations des méthodes et pratiques des développeurs,

afin d'en tirer des conclusions et, *in fine*, de proposer des améliorations. Pour construire cette connaissance empirique, les dépôts de code sont des points d'entrée privilégiés puisqu'ils permettent non seulement l'accès des projets nombreux et variés mais plus encore à l'historique de leurs versions, fournissant ainsi des possibilités d'analyse plus nombreuses, dynamiques et précises. Un aspect du génie logiciel empirique concerne la construction de mesures quantitatives pour la qualité des logiciels. Ces mesures appelées métriques sont utilisées par les développeurs et les décideurs des projets afin de qualifier intrinsèquement ou extrinsèquement la qualité d'un logiciel. Certaines mesures sont focalisées sur le code. Nous pouvons par exemple citer le nombre de McCabe [McC76] qui mesure la complexité du code produit. D'autres métriques utilisent des éléments indirectement liés au code. Parmi ces éléments nous retrouvons les différentes méta-données énumérées plus haut. Connaître le nombre de tickets de bogue peut, par exemple, être un indicateur de la qualité du logiciel produit. En effet, un nombre élevé de tickets ouverts peut indiquer un problème sur la qualité du logiciel.

À l'aide de ces métriques, la construction de systèmes prédictifs utilisant de l'intelligence artificielle et dédiés au génie logiciel est rendue possible. Ces systèmes utilisent des algorithmes d'apprentissage automatique sur des données issues des dépôts de code ou encore sur des métriques logicielles. Les systèmes d'apprentissage automatiques peuvent être supervisés ou non-supervisés. Les systèmes supervisés utilisent un résultat connu par avance afin d'ajuster leurs paramètres et minimiser l'erreur de prédiction. Les systèmes non-supervisés apprennent sur les données sans connaître le résultat par avance. Ils permettent, par exemple, le regroupement de données en fonction de leurs caractéristiques.

Ces indicateurs de qualité (métriques) et les systèmes prédictifs peuvent s'appliquer à divers aspects du logiciel. Leur combinaison permet de créer des outils assistant la prise de décisions dans les projets. L'aspect sur lequel nous nous focalisons dans cette thèse concerne le développement des architectures logicielles et les développeurs y contribuant. L'architecture définit la structuration interne du logiciel. L'expert en architecture logicielle est communément appelé architecte et se distingue par une pluralité de connaissances et de compétences, ainsi qu'une bonne capacité d'abstraction. Les architectes sont souvent les développeurs les plus expérimentés dans les projets [Kru99, McB07]. Identifier ces experts permet une meilleure gestion de projet. Or, cette expertise n'est pas explicite dans les méta-données du projet. La caractérisation automatisée de ces développeurs experts est donc une solution qui permet, par exemple, d'anticiper le départ des projets de ressources critiques et de minimiser ainsi la perte de connaissances et de compétences sur l'architecture. Ainsi, ces systèmes de mesure et de prédiction viennent en soutien aux gestionnaires et architectes du projet.

## 1.2 Contributions

Les tickets décrivent des différentes demandes qui surviennent tout au long du cycle de vie d'un logiciel. Il peut s'agir par exemple d'une demande de *refactoring*, de développement d'une nou-

velle fonctionnalité, de documentation ou encore de correction d'un bogue. C'est sur cette dernière catégorie de tickets que nous focalisons nos travaux. En effet, le nombre de bogues peut-être un indicateur pour la conduite de projet, en particulier ceux liés à l'architecture logicielle. C'est pourquoi nous avons mis au point un classifieur automatique de tickets de bogues dont les performances, validées sur un jeu de données de référence, dépassent celle de l'état de l'art. Nos travaux permettent de répondre à la Question de recherche 1.

**Question de recherche 1**

Est-il possible d'identifier de manière automatique, parmi l'ensemble des tickets d'un projet, les tickets décrivant des bogues, dans le but d'utiliser cette identification comme indicateur de qualité ?

Mesurer la contribution des développeurs à l'écriture d'un logiciel permet notamment l'identification des développeurs participant de manière significative au processus de développement. Cette mesure et la catégorisation des développeurs a été décrite pour la première fois par Bird *et al.* [BNM+11]. Ce type de mesure peut servir pour la conduite de projet. En effet, savoir évaluer la contribution des développeurs permet d'identifier ceux ayant acquis une expérience par le biais de leurs contributions au projet. Cependant, il n'existe pas de métrique dédiée à la mesure de la contribution architecturale. Une des propositions de cette thèse est l'étude d'une métrique d'évaluation de la contribution architecturale dérivant d'une métrique d'évaluation de la contribution globale au développement d'une application. Nous proposons son évaluation sur 13 versions d'un projet logiciel *open-source*. Enfin, nous faisons une catégorisation des développeurs et observons leur *turn-over*. Cette contribution répond à la Question de recherche 2.

**Question de recherche 2**

Est-il possible de créer une métrique d'évaluation de la contribution à l'architecture *runtime* ? Sur la base de celle-ci, est-il possible de catégoriser les développeurs ?

Les développeurs expérimentés d'un projet sont souvent ceux portant la connaissance historique mais aussi technique du projet. Cette connaissance technique des développeurs expérimentés recouvre souvent celle de l'architecture logicielle. La perte de connaissances, induite par le départ de ces personnes d'un projet, n'est pas négligeable et peut être source de problématiques de gestion des ressources humaines et techniques. L'identification automatisée de ces développeurs permet une meilleure gestion des équipes de développement. Dans cette thèse, nous proposons un système automatisé de détection des développeurs expérimentés utilisant un ensemble de métriques logicielles et un classifieur entraîné par apprentissage supervisé sur un jeu de données que nous avons construit. Cette partie de la thèse répond à la Question de recherche 3.

**Question de recherche 3**

Est-il possible de détecter, à l'aide d'un système de classification, les développeurs expérimentés sur la base de métriques logicielles ?

Croiser nos travaux sur la mesure de la contribution à l'architecture et ceux sur la classification des développeurs expérimentés nous permet ensuite d'étudier le profil des développeurs expérimentés détectés par notre classifieur en matière de contributions architecturales. Un deuxième objectif de ce croisement des approches est d'estimer leur validité. Il est en effet attendu que les développeurs expérimentés soient des contributeurs majeurs au développement en général et à l'architecture en particulier. Cette contribution répond à la Question de recherche 4.

**Question de recherche 4**

Quel est le profil des développeurs expérimentés en termes de contributions architecturales ?

Ces questions de recherche générales seront affinées dans les chapitres présentant les contributions.

## 1.3 Plan de la thèse

Ce manuscrit de thèse est organisé comme suit :

- La Partie I situe le contexte de cette thèse et présente l'état de l'art.
  - Le Chapitre I.2 détaille le contexte de cette thèse. Il introduit les concepts associés aux architectures logicielles et plus spécifiquement aux architectures *runtime*. Il présente ensuite la notion de mesure de la qualité associée aux logiciels ou aux processus de développement à l'aide de différents types de métriques. Puis, il aborde l'évaluation de l'expertise des développeurs et plus spécifiquement celle des architectes. Il explique enfin les différents types d'apprentissage automatiques évoqués dans cette thèse.
  - Le Chapitre I.3 présente l'état de l'art de cette thèse. Il est divisé en trois parties. La première présente les travaux liés à la classification de tickets. La seconde porte sur les métriques d'évaluation des contributions logicielles. Enfin, la troisième se focalise sur les approches de regroupement et de profilage des développeurs.
- La Partie II présente les différentes contributions de cette thèse.
  - Le Chapitre II.4 présente une approche de classification automatique de tickets de bogues. Il détaille la constitution du jeu de données de 5591 tickets, les étapes de traitement automatique du langage naturel utilisées pour transformer les tickets en vecteurs de valeurs numériques utilisables dans un processus d'apprentissage machine, la sélection d'un classifieur supervisé et son optimisation. Ce chapitre présente également une

analyse des performances obtenues par le classifieur. Enfin, une explication de la classification au travers de l'importance des termes contenus dans les tickets est proposée (grâce à des techniques d'*Explainable AI*).

- Le Chapitre II.5 présente une approche permettant de catégoriser les développeurs en fonction de leurs contributions au développement de l'architecture *runtime*. Il définit un modèle formel de mesure de la contribution au développement de l'architecture *runtime* et la catégorisation des développeurs associée. L'approche est expérimentée sur l'historique complet d'un projet extrait de GitHub. Les résultats obtenus sur la catégorisation des développeurs valident cette preuve de concept.
- Le Chapitre II.6 présente une approche de classification automatique des développeurs expérimentés. Il détaille la construction du jeu de données de 703 développeurs à partir de seize projets *open-source*. Il introduit ensuite le traitement des données, la sélection d'un classifieur supervisé et la génération de données synthétiques permettant de compenser le déséquilibre des données. Ce chapitre présente ensuite les résultats obtenus et fournit une analyse des variables influençant le classifieur dans ses choix (grâce à des techniques d'*Explainable AI*).
- Le Chapitre II.7 présente enfin une étude du profil des développeurs expérimentés, en termes de contributions au développement architectural, à l'aide d'un croisement des approches proposées au Chapitre II.5 et au Chapitre II.6.

- Le Chapitre 8 décrit les mesures de diffusion du code et des données produits dans cette thèse et évalue la reproductibilité de nos travaux.
- Enfin, le Chapitre 9 conclut avec un résumé des contributions de cette thèse, discute leurs limitations et présente les perspectives de ces travaux.

# Première partie

# Contexte de l'étude

# Chapitre I.2

# Assistance à la prise de décisions architecturales

## Sommaire

Comme introduit dans le Chapitre 1, cette thèse s'inscrit dans le cadre de l'assistance à la prise de décisions des architectes et gestionnaires de projets par le biais d'indicateurs (métriques) et de méthodes prédictives. Ce chapitre présente le positionnement scientifique et technique dans lequel s'inscrit cette thèse.

## 2.1 Concepts de base des architectures logicielles

### 2.1.1 Composants logiciels

Un aspect fondamental d'une architecture logicielle est la modularité offerte par le découpage de l'application en différentes unités, qui assemblées, forment un tout. Ces unités qui composent l'architecture logicielle sont souvent appelées composants. La définition du concept de composant a fini par atteindre une forme de consensus. Une première définition est donnée par Taylor [TMD09] :

> *"Un composant logiciel est une entité architecturale qui (1) encapsule un sous-ensemble de fonctionnalités du système et/ou des données, (2) restreint l'accès au sous-ensemble via une interface explicitement définie, et (3) qui a des dépendances requises explicitement définies dans le contexte d'exécution."*

Cette définition de Taylor donne une vision explicite et claire de ce qu'est un composant. Celui-ci satisfait des propriétés de rassemblement des données et des fonctionnalités dans une structure (encapsulation), de limitation volontaire de l'utilisation du composant par le biais de son interface mais aussi l'explicitation des dépendances qui lui sont nécessaires pour fonctionner.
Szyperski [SGM02] donne une définition similaire en considérant le composant comme une unité logicielle dont l'implémentation et le fonctionnement sont masqués. Seules sont visibles les interfaces requises et fournies :

> *"Un composant logiciel est une unité de composition ayant des interfaces contractuellement spécifiées et uniquement des dépendances explicites. Un composant logiciel peut-être déployé indépendamment et est sujet à composition par les tierces parties."*

En utilisant ces deux définitions nous pouvons dire qu'un composant est une unité logicielle fonctionnellement indépendante exposant des interfaces permettant la composition logicielle. Les interfaces exposées sont de deux types : requises et fournies. Les interfaces fournies mettent à disposition des services implantés par un composant et pouvant être utilisés par d'autres composants. Les interfaces requises permettent à un composant d'invoquer les services dont son fonctionnement dépend et qui lui sont fournis par d'autres composants externes. Les systèmes d'interfaces permettent ainsi un découplage des composants. De cette manière, les unités logicielles sont réutilisables dans d'autres contextes et applications.

### 2.1.2 Connecteurs entre composants

Un autre aspect fondamental des architectures concerne les connecteurs entre composants. Ces connecteurs sont destinés à lier les composants entre-eux afin qu'un composant puisse invoquer un

service d'un autre composant et vice-versa [LB20]. D'après Taylor *et al.* [TMD09], les connecteurs sont les médiateurs qui connectent les composants et permettent ainsi leurs interactions. Ainsi, la partie calcul et algorithmique est gérée par le composant tandis que les interactions sont gérées par les connecteurs, permettant un meilleur découplage dans l'application.

## 2.2 Niveaux d'architecture

L'architecture logicielle est au cœur de la construction et de l'évolution des systèmes [Pre97, TMD09]. Elle consiste en une structuration dont va dépendre l'organisation d'un logiciel et de ses composants. Une architecture peut se décliner en différents niveaux du plus concret (instances de composants) au plus abstrait avec, par exemple, les patrons architecturaux.

### 2.2.1 Le modèle d'architecture Dedal

Pour gérer les architectures logicielles, Zhang *et al.* [ZUV10, ZZU+12] ont proposé un modèle d'architecture à 3 niveaux, le modèle d'architecture Dedal. Ce modèle permet de séparer clairement trois niveaux d'abstraction : spécification, configuration et assemblage. La Figure I.2.1 illustre les différents concepts définis dans Dedal. Cet exemple d'une petite architecture logicielle a été repris des travaux de Mokni *et al.* [MHU+15]. L'architecture décrite est est celle d'un logiciel de domotique contrôlant automatiquement l'éclairage du bâtiment en fonction de l'heure. Un composant `Orchestrator` interagit avec les appareils appropriés pour jouer le scénario souhaité.

#### 2.2.1.1 Le niveau spécification de l'architecture

Le niveau spécification permet d'expliciter les exigences fonctionnelles du logiciel. Son but est de donner une vue abstraite des éléments logiciels impliqués (composants). A ce niveau d'architecture, les décisions de conception consistent à identifier les types de composants abstraits qui seront utilisés pour réaliser les exigences fonctionnelles. Ces composants abstraits sont appelés rôles de composants. Les rôle des composants sont destinés à déclarer l'ensemble des fonctionnalités attendues des composants par la spécification d'interfaces. Les rôles de composants sont ainsi le support des processus de recherche et de sélection des composants concrets permettant d'implémenter l'architecture.
La Figure I.2.1 (a) donne l'exemple de la spécification du *Home Automation System* (HAS). Celui-ci est constitué du composant `HomeOrchestrator`, qui gère l'éclairage en utilisant les rôles de composants `Light` et `Luminosity`, ainsi que le rôle de composant `Time`.

#### 2.2.1.2 Le niveau configuration de l'architecture

Le niveau configuration décrit l'implémentation d'une architecture. Une configuration est définie par l'ensemble des composants qui correspondent le mieux aux rôles définis au niveau spécification. Ces composants sont appelés classes de composants. Une classe de composants correspond à un

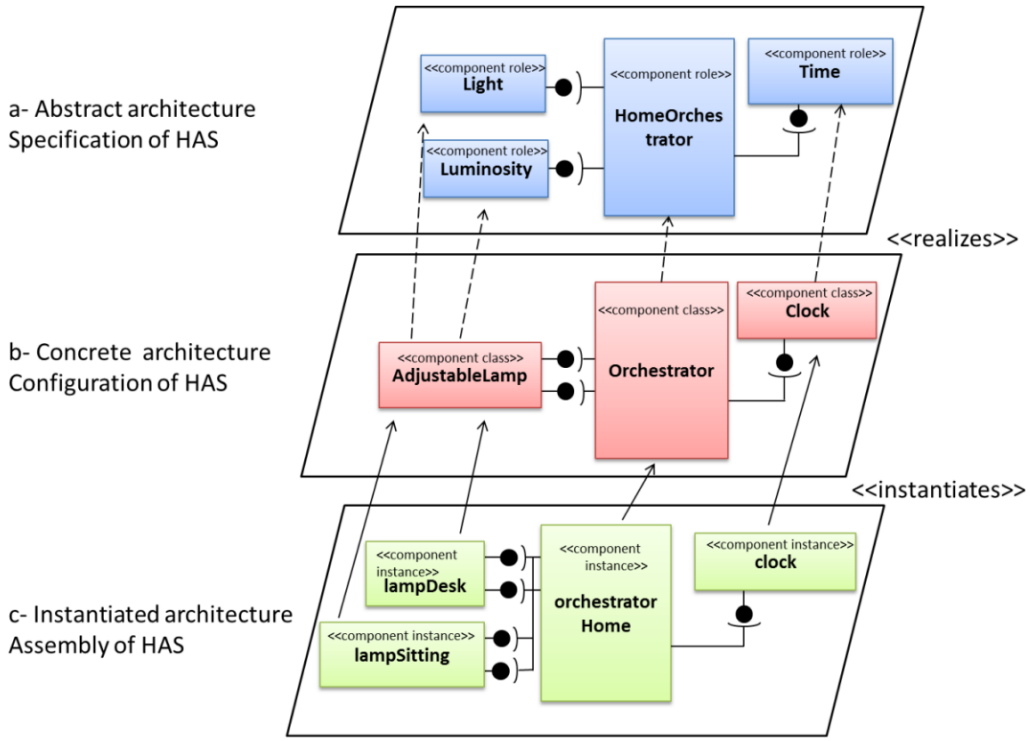



FIGURE I.2.1 – Niveaux d'architecture de Dedal pour le système HAS [MHU+15, LB20]

.

composant logiciel dont le code est disponible dans un dépôt. Chaque classe de composants sélectionnée pour construire une configuration est l'implémentation d'un rôle défini dans la spécification correspondante (le type de la classe de composants spécialise le type du rôle associé).

La Figure I.2.1 (b) montre une implémentation (configuration) du HAS conforme à la spécification. Dans cette configuration, `Orchestrator` réalise `HomeOrchestrator`, `Clock` réalise `Time`, tandis que `AdjustableLamp` réalise les rôles des composants `Light` et `Luminosity`.

#### 2.2.1.3 Le niveau assemblage de l'architecture

Le niveau assemblage correspond au déploiement de l'architecture lors de son exécution. Ce niveau est visible sur la Figure I.2.1 (c). Les composants du niveau assemblage sont appelés instances de composants. Le niveau assemblage décrit la liste des instances de composants et leurs contraintes d'assemblage, comme les cardinalités des connections. Il décrit également les valeurs d'initialisation permettant d'adapter les différentes instances de composants à des besoins ou des contextes d'exécution spécifiques.

### 2.2.2 Patrons architecturaux

Une seconde manière d'abstraire une architecture logicielle est d'en avoir une vision au travers du prisme des patrons architecturaux. Les patrons architecturaux permettent de répondre à des problématiques de conception grâce à des "formats" architecturaux de référence. Cette standardisation de l'architecture permet une adaptation à un grand nombre de problématiques architecturales récurrentes : création d'instances uniques, création d'interfaces utilisateurs, structures arborescentes, etc. Les patrons architecturaux n'ayant pas de rôle spécifique au sens de ceux définis dans Dedal (voir Section 2.2.1.1), ils ont un niveau d'abstraction encore plus élevé que le niveau de spécification de Dedal. Parmi les principaux patrons de conception nous pouvons citer le modèle-vue-contrôleur, le singleton, la stratégie, etc.

#### 2.2.2.1 Styles architecturaux

Les styles architecturaux sont des principes d'architecture avec un haut niveau d'abstraction et une grosse granularité. Selon Garlan et Shaw [GS93], un style architectural va énoncer et déterminer le vocabulaire des différents composants et connecteurs qui peuvent être utilisés. Le style décrit également l'ensemble des contraintes [THK07, TSTD16] pouvant s'appliquer sur ces derniers. Les contraintes peuvent concerner la topologie des composants, en interdisant par exemple les cycles entre les composants. Ces contraintes peuvent aussi s'appliquer à la sémantique du style. Les styles architecturaux permettent ainsi de fournir un langage commun dont la sémantique est clairement définie et qui s'abstrait des technologies mettant en œuvre le style. Notons que l'on peut trouver au sein d'une application plusieurs styles architecturaux. Parmi les styles architecturaux de référence, nous pouvons citer *Service-Oriented Architecture*, N-tiers, client/serveur, etc.

#### 2.2.2.2 Patrons architecturaux

Les patrons architecturaux sont des solutions architecturales génériques permettant de répondre à des problématiques de conception récurrentes [GS93]. Ils décrivent une solution standard de conception ayant une portée plus large et plus abstraite que les patrons de conception logiciel. Ces patrons architecturaux ne sont pas orthogonaux aux styles architecturaux, un patron architectural peut s'inscrire dans l'utilisation d'un style. Parmi les patrons architecturaux de référence nous pouvons citer le patron *broker*, le patron singleton, le patron Modèle-Vue-Contrôleur, le patron *Pipes and Filters*, le patron Blackboard, etc.

### 2.2.3 Architecture à l'exécution

L'architecture à l'exécution (*runtime architecture*) est un niveau architectural particulier correspondant à l'ensemble des éléments structurels (objets, composants par exemple en fonction des paradigmes) qui sont instanciés lors du lancement d'une application pour initier ou supporter son exécution. Selon le modèle d'architecture Dedal (voir Section 2.2.1), ce type d'architecture se place

au niveau assemblage (voir Section 2.2.1.3). De manière *ad hoc*, l'architecture *runtime* est classiquement définie par le code du programme principal et est déployée lors de l'exécution de ce dernier. Pour faciliter le développement et la maintenance des architectures *runtime*, il est possible d'utiliser des *frameworks* proposant des mécanismes génériques d'instanciation de l'architecture *runtime*. Ces mécanismes sont en général proposés sous la forme d'un conteneur fournissant des fonctionnalités de création et de connexion de composants basées sur l'inversion de contrôle et l'injection de dépendances. L'architecture à déployer peut être explicitement définie par un descripteur de déploiement écrit à l'aide d'un DSL et interprété par le conteneur (cas du *framework* Spring étudié dans cette thèse). Elle peut également implicitement résulter des dépendances définies par les composants qui sont dynamiquement résolues lors de leur instanciation par le conteneur (cas des frameworks OSGI, JEE ou de Spring [AA05, SME+17]). Il s'agit du type d'architectures *runtime* que nous avons étudié dans cette thèse, les architectures "modernes", outillées. Des mécanismes spécifiques permettent l'instanciation des composants et la création des liens les reliant lors de l'exécution. Nous verrons dans cette section les avantages majeurs de ce genre d'architecture comparativement aux architectures statiques. Ces avantages proviennent de leur aspect déclaratif, de la séparation des préoccupations mais aussi de la facilité et de la souplesse de création des architectures [PLBUV19].

#### 2.2.3.1 Inversion de contrôle

Le terme d'inversion de contrôle (*Inversion of Control* ou *IoC*) a été utilisé pour la première fois dans la thèse de Mattsson en 1996 [Mat96]. Ce mécanisme a ensuite été popularisé par Fowler en 2004 [Fow04]. L'inversion de contrôle désigne le fait d'inverser le flot d'exécution du programme. Ce n'est plus le programme qui contrôle l'exécution mais un *framework* dédié. Il se charge d'instancier les composants constituant l'architecture runtime de l'application et de satisfaire leur dépendances. Ainsi, le code des composants peut se concentrer sur la logique métier et devenir générique, portable entre différents environnements d'exécutions. Le code technique mettant en oeuvre les ressources éventuellement spécifiques de l'environnement est fourni et géré par le *framework*.

#### 2.2.3.2 Injection de dépendances

L'injection de dépendances est une forme d'inversion de contrôle. Le *framework* gérant l'exécution assure l'instanciation automatique des composants déclarés par l'application mais également la satisfaction de leurs dépendances. Les composants déclarent leurs dépendances sous la forme d'une interface requise. Le *framework* recherche, dans le contexte d'exécution, les composants ou les ressources permettant de satisfaire ces dépendances et crée les liens nécessaires (initialisations). Les liens de dépendance ne sont ainsi ne sont plus déterminés statiquement dans le code de l'application mais dynamiquement à l'exécution par le *framework*.
Le Listing I.2.1 donne un exemple d'instanciation directe de composants. Dans cet exemple, les relations entre les composant sont statiques. L'objet `Orchestrator` et ses dépendances `Clock` et `Lamp` sont définis avant l'exécution par le développeur. Le Listing I.2.2 reprend l'exemple du Lis-

ting I.2.1 mais en l'adaptant à l'inversion de contrôle. Dans le Listing I.2.2, c'est le *framework* d'exécution qui, détectant la dépendance déclarée à l'aide de l'annotation `@Inject`, crée les instances de `Clock` et `Lamp` et les injectent dans l'objet `Orchestrator` par invocation du constructeur associé à l'annotation `@Inject`.

```
public class Orchestractor{
  public Clock clockDep;
  public Lamp lampDep;

  public Orchestrator(){
    clockDep = new Clock();
    lampDep = new Lamp();
  }
}
```

Listing I.2.1 – Instanciation directe (statique).

```
public class Orchestractor{
  public Clock clockDep;
  public Lamp lampDep;

  @Inject
  public Orchestrator(Clock clock,Lamp lamp){
    clockDep = clock;
    lampDep = lamp;
  }
}
```

Listing I.2.2 – Injection de dépendances (dynamique).

Ekstrand et Ludwig [EL16] voient plusieurs avantages à l'injection de dépendances :

- Une réduction drastique du couplage liée au fait que le *framework* se charge des connexions entre composants.
- Les composants n'ont plus la connaissance explicite de leurs dépendances. Ils ne connaissent pas par avance les instances qui leur seront fournies pour satisfaire leurs dépendances mais savent juste que ces dernières seront satisfaites.
- Les composants sont plus facilement reconfigurables par changement de l'implémentation de leurs dépendances sans modification des composants eux-mêmes.
- Les systèmes basés sur l'injection de dépendances sont plus facilement maintenables grâce à un découplage supérieur mais aussi parce que les interactions des composants avec le systèmes sont automatiquement définies par le *framework* gérant l'injection.

### 2.2.4 Framework Spring

Un des *frameworks* implémentant l'inversion de contrôle et l'injection de dépendances est Spring [1]. Spring est un *framework* Java créé en 2002 par l'entreprise Pivotal Software puis racheté par VMWare. Spring est un initialement dédié à la conception d'architectures *runtime* permettant la création de services web. Les services web créés avec Spring se basent sur le patron de conception Modèle-Vue-Contrôleur (MVC). Spring est l'un des *frameworks* les plus utilisés pour la création de services web [Map16]. Spring fournit trois moyens pour créer des architectures *runtime* qui peuvent être combinés [PLBUV19] :
Le moyen historique implémenté par Spring est celui illustré par le Listing I.2.3. Les composants nécessaires à l'application sont décrits dans un fichier XML appelé descripteur de déploiement. Les composants à instancier sont décrits par des balises `bean` où sont définis la classe à utiliser et l'identifiant du composant. Les liens entre composants sont réalisés à l'aide de l'identifiant de

1. https://spring.io/

chacun. Dans le Listing I.2.3, le composant `Orchestrator` utilise les balises `property` pour définir ses dépendances (liens nécessaires vers d'autres composants).
Le second moyen mis à disposition par Spring pour la création d'architectures est un fichier de configuration Java spécifique comme montré par le Listing I.2.4. Cette méthode est le pendant du descripteur de déploiement XML. À la place des balises XML, des annotations Java sont utilisées pour définir les composants (`@Bean`).

```
<beans xmlns="[... schema Spring ...]">
  <bean class="my.smartHome.Clock"
       id="clock1"/>
  <bean class="my.smartHome.Lamp"
       id="lamp1"/>

  <bean class="my.smartHome.Orchestrator"
       id="myOrchestrator">
         <property name="time" ref="clock1"/>
         <property name="light" ref="lamp1"/>
   </bean>
</beans>
```

Listing I.2.3 – Descripteur de déploiement XML

```
@Configuration
public class BeansConfiguration{
  @Bean
  public Clock clock1(){return new Clock();}
  @Bean
  public Lamp lamp1(){
    return new AdjustableLamp();
  }
  @Bean
  public Orchestrator myOrchestrator(){
    return new
         Orchestrator(clock1(),lamp1());
  }
}
```

Listing I.2.4 – Fichier de configuration Java

Un troisième moyen de créer des architectures *runtime* avec Spring est d'utiliser une configuration auto-câblée. Ce type de configuration est illustré par les Listings I.2.5 et I.2.6. Dans le Listing I.2.6, `Ochestrator` est un composant requérant deux dépendances : `Lamp` et `Clock`. Pour satisfaire les dépendances de manière automatique, les dépendances du composant `Ochestrator` sont annotées avec `@Autowired`. Quand le *framework* détecte *runtime* la présence de `@Autowired`, il recherche automatiquement dans le contexte d'exécution les composants qui peuvent satisfaire ces dépendances.
Pour cela, le *framework* repère tous les `@Bean` / `<bean>`, instancie les composants correspondants et les enregistre dans le contexte d'exécution (assimilable à un registre de composants). Quand un `@Autowired` est rencontré, il fait une recherche dans le contexte et connecte le composant disponible, s'il existe, à moins qu'il y ait une situation ambiguë (plusieurs candidats possibles). Dans ce dernier cas, il faut avoir recours à des critères de filtrage soit en donnant priorité à un composant, soit par type, soit par nom. Dans notre cas, `Clock` et `Lamp` ont été annoté avec `@Component` dans le Listing I.2.5. Spring a donc instancié et enregistré ces composants dans le contexte d'exécution. Il peut donc attacher une instance de `Clock` et de `Lamp` à `Ochestrator`.

## 2.3 Métriques logicielles

Le logiciel est un produit particulier car immatériel. Cette particularité le rend à la fois puissant et vulnérable. Il est puissant car configurable et adaptable. Vulnérable car potentiellement complexe, dépendant d'un environnement d'exécution (logiciel et matériel). Tout comme pour un objet physique, dont les rouages peuvent être observés, le code logiciel obéit à une logique similaire. Il est

```
@Component
public class Clock implements Time {/*...*/}

@Component
public class Lamp implements Light {/*...*/}
```

Listing I.2.5 – Configuration auto-câblée, composants Clock et Lamp.

```
@Component
public class Orchestrator{
  @Autowired
  private Time time;
  @Autowired
  private Light light;
}
```

Listing I.2.6 – Configuration auto-câblée, composant Ochestrator.

possible d'évaluer la qualité d'un logiciel en utilisant des mesures s'appuyant sur différentes caractéristiques. Ces mesures appliquées au monde du logiciel s'appellent les métriques. Les premières métriques se sont concentrées sur le code logiciel. Elles sont appelées *code metrics*. Cependant, avec l'invention des systèmes de gestion de versions et la création des chaînes de production logicielles, un nouveau type de métriques est apparu : les métriques de processus ou *process metrics*. Ces nouvelles métriques permettent la mesure d'éléments indirectement liés au code des logiciels.

### 2.3.1 Métriques liées au code

Les métriques liées au code (*code metrics*) se répartissent en trois catégories : les métriques de complexité, les métriques dédiées à la programmation orientée objet et les métriques permettant de mesurer la maintenabilité.

#### 2.3.1.1 Métriques de complexité

En 1976, McCabe [McC76] propose la première métrique dédiée au logiciel : la complexité cyclomatique ou nombre de McCabe. Cette métrique permet de mesurer la complexité du code au travers du graphe de flot de contrôle d'un programme. Le nombre de McCabe mesure le nombre de chemins linéairement indépendants dans le graphe, permettant ainsi de mesurer la complexité des décisions algorithmiques.
Il est suivi de près par Halstead en 1977 [Hal77] qui propose un ensemble de métriques permettant de mesurer le volume du programme, le niveau de difficulté du programme, l'effort à l'implémentation, le niveau du programme (inverse du niveau de difficulté), le temps d'implémentation et une estimation du nombre de bogues potentielles. Le calcul de ces métriques est basé sur des jetons représentant les opérateurs et les opérandes du programme.

#### 2.3.1.2 Métriques dédiées à la programmation orientée objet

Avec l'avènement de la programmation orientée objet (POO) dans les années 90, il devient nécessaire d'avoir des outils de métrologie logicielle dédiés à la POO. En 1994, Chidamber et Kemerer [CK94] inventent une suite de métriques spécifiquement conçues pour la POO. Ces métriques couramment appelées métriques de CK ou *CK-metrics* sont encore aujourd'hui des références. Les métriques de CK permettent de mesurer la complexité du code dédié à la POO grâce à cinq types

de mesures : le poids des méthodes par classe, la profondeur d'héritage, le nombre d'enfants pour une classe, le couplage entre classes et le manque de cohésion des méthodes.
En 1995, Abreu *et al.* [AGE95] ont également inventé une suite de métriques nommée MOOD (*Metrics for Object Oriented Design*) dédiées à la POO. Abreu *et al.* ont proposé des métriques permettant d'évaluer les éléments suivants : l'encapsulation, le couplage, le polymorphisme, le pourcentage de méthodes héritées.
Martin *et al.* [MNK03] proposent en 2003 des métriques dédiées aux *packages*. Les métriques de Martin *et al.* mesurent : le nombre de classes et d'interfaces dans les *packages*, le niveau d'abstraction des *packages*, leur instabilité (résilience aux changements des *packages*) et les dépendances cycliques entre *packages*. Martin *et al.* mesurent également le couplage afférent (le nombre de classes hors d'un *package* dépendant des classes du *package*) et le couplage efférent (le nombre de classes d'un *package* dépendants de classes d'un autre *package*).

#### 2.3.1.3 Métriques de maintenabilité

Des métriques dédiées au code et de celles liées à la POO ont dérivé des métriques permettant de mesurer la maintenabilité d'un logiciel. Oman et Hagemesiter [OH92] ont proposé en 1992 un panel de métriques permettant de mesurer notamment l'intensité de la maintenance (temps en heures) à effectuer sur le code, l'intensité des défauts (gravité) ainsi que le taux de réutilisation du code dans d'autres projets. Microsoft [HJ21] a développé un indice de maintenabilité intégré à l'éditeur Visual Studio. Le calcul de cet indice est réalisé en utilisant les métriques de Chidamber et Kemerer [CK94] mais aussi celles de Halstead [Hal77].

### 2.3.2 Métriques et *smells* dédiées à l'architecture

L'invention des concepts d'architectures à composants [Pre97, SGM02] et des principes de POO a amené la communauté scientifique à trouver des métriques permettant d'évaluer la qualité des architectures logicielles. Ces métriques s'appuient à la fois sur les travaux liés aux métriques du code mais aussi sur les *code smells* architecturaux. Le terme de *code smell* a été employé pour la première fois par Beck et Fowler [BFB99] pour désigner une portion de code ou de programme apparaissant comme une mauvaise solution et intuitivement repérable à la manière d'une mauvaise odeur. Beck et Fowler dans "Refactoring: Improving the design of existing code" [BFB99] ont décrit 22 *bad smells*. Notons cependant qu'ils n'ont fait que décrire des *code smells* mais n'ont pas défini de manière quantifiable comment les détecter. De plus, ces *bad smells* ne tiennent pas compte de l'architecture *runtime*.
Ce n'est qu'en 2015 que Fontana *et al.*, en s'appuyant pour partie sur les métriques de [CK94] mais aussi sur les *bad smells* de Beck et Fowler [BFB99], ont réussi à définir de manière empirique des seuils permettant de détecter des *bad smells* architecturaux.
Puis, en 2018, Aniche *et al.* [ABT+18] ont établi une liste de six *code smells* dédiés aux architectures utilisant le patron MVC comme celles pouvant être créées avec Spring. La même année, Roveda *et*

*al.* [RFPZ18] ont présenté une métrique permettant de mesurer la dette technique architecturale en se basant sur des métriques liées au code, au processus mais également sur des *code smells* de Beck et Fowler [BFB99].

### 2.3.3 Métriques liées au processus

Les métriques liées au processus ou *process metrics* ne sont pas définies de manière précise dans la littérature. Ces métriques prennent leurs sources au début des années 2000 puis se sont progressivement démocratisées dans le domaine logiciel. Elles ont suivi l'évolution de la programmation mais surtout des systèmes et outils entourant la création de code logiciel. Ces systèmes sont porteurs d'informations autour du code (méta-données). Ces méta-données peuvent pour certaines être utilisées pour construire des métriques de processus. Des sources principales de métriques de processus sont les systèmes de gestion de versions (Git, SVN, etc.), les plates-formes d'hébergement (GitLab, GitHub, SourceForge, etc.) ou celles dédiées à l'intégration continue (Travis, Jenkins, etc.). Nous pouvons citer, par exemple, le nombre de développeurs participant à un projet logiciel, l'âge du projet, le nombre de *commits*, la fréquence des changements, le nombre de versions, le nombre de constructions réussies, etc.

## 2.4 Expertise en développement logiciel

Le concept d'expertise en développement logiciel n'est pas récent. Il apparaît lors la démocratisation de l'informatique dans les années 80. En 1981, Jeffries *et al.* [JTPA81] ont comparé deux groupes de développeurs. Un premier groupe est constitué d'étudiants considérés comme inexpérimentés tandis que le second est constitué de développeurs professionnels. Suite à cette étude, Jeffries *et al.* ont conclu que l'expérience acquise par les développeurs permettait de réduire le temps requis pour développer les fonctions d'un programme. Cependant dès le milieu des années 90, Sonnentag [Son95, Son98] a montré que la simple dimension temporelle n'explique pas forcément le caractère expérimenté d'un développeur. Sur la base de ces travaux Sonnentag *et al.* [SNV06] font la différence entre l'expertise basée sur le temps et l'expertise relevant d'une haute performance sur un domaine. Sur un aspect plus psychologique, Ericsson [Eri06] définit l'expertise comme :

> *"les caractéristiques, compétences et connaissances qui distinguent les experts des novices et des personnes moins expérimentées."*

Cette définition, bien que très globale, tient compte des caractères spécifiques (aptitudes particulières, compétences, connaissances) qui font l'expertise du développeur, confortant ainsi les travaux de Sonnetag [Son95, Son98, SNV06]. En 2014, Siegmund *et al.* [SKL+14] ont empiriquement mesuré l'expérience des développeurs. Ils notent l'absence de définition précise de l'expérience du développeur. Dans leurs discussions avec des experts techniques, Siegmund *et al.* relèvent des divergences mais également des points communs à la notion d'expérience. C'est sur la base de ces discussions que Siegmund *et al.* définissent l'expérience comme suit :

> *"L'expérience en programmation décrit la quantité de connaissances acquises en matière de développement de programmes, de sorte que la capacité à analyser et à créer des programmes est améliorée."*

Siegmund *et al.* signalent que l'expérience n'est ni plus ni moins que la somme des connaissances apprises et de ses réalisations. Cependant, elle est à distinguer du fait d'être doué ou non pour programmer. Ainsi les travaux de Siegmund *et al.* rejoignent ceux de Sonnetag *et al.* [SNV06] sur la dissociation d'une expertise quantitative ou qualitative. Notons que ces deux acceptions de l'expertise peuvent être mesurées par des métriques.

## 2.5 Expertise et rôle d'un architecte

Avec l'émergence de l'approche par composants ou *component-based software engineering* (CBSE) à la fin des années 90 [Pre97], la spécialisation des développeurs entraîne la création d'un nouveau rôle, celui d'architecte logiciel. Tout comme l'expertise, la définition même de l'architecte est parfois soumise à des divergences. Ceci est d'autant plus vrai que la notion d'architecte fait référence à l'expertise, à l'expérience et aux compétences non-techniques spécifiques à certains développeurs. La première définition d'architecte que nous pouvons citer est celle de Booch [Boo96] :

> *"Chaque projet devrait avoir exactement un architecte identifiable et pour les projets de grande envergure, l'architecte principal doit être assisté d'une équipe d'architectes de taille modeste."*

Cette phrase souligne l'importance et le rôle clef de l'architecte dans un projet. McBride [McB07] met en lumière l'aspect technique dans sa définition de l'architecte en le mentionnant comme :

> *"responsable du design et des choix technologiques dans le processus de développement logiciel."*

La définition donnée par McBride englobe à la fois l'expertise technique mais également la prise de décisions tout au long du cycle de vie du logiciel. Cela montre l'aspect global des compétences et connaissances dont doit disposer l'architecte. En effet, la notion d'architecte s'arrête souvent à la seule spécificité d'expert technique. C'est pourquoi McBride insiste sur le fait que l'architecte a un panel de compétences vaste allant de la conception globale à la gestion des composants d'un système logiciel. Cette navigation entre les différents niveaux du système tout au long de son cycle de vie est montrée par la Figure I.2.2.
La Figure I.2.2 montre la complexité de la tâche de gestion d'un projet durant son cycle de vie. Elle montre également l'abstraction et la concrétisation dont doit faire preuve l'architecte. Les aptitudes non-techniques requises pour gérer ces aspects ont été mises en lumière pour la première fois par Krutchen [Kru99]. Plusieurs autres travaux [McB07, Ber08, SH15] ont ensuite listé les différents aspects managériaux et humains dont doit faire preuve l'architecte. Ces différents aspects sont aujourd'hui régulièrement appelés *soft-skills* et ne sont pas directement mesurables par le biais du code ou des méta-données comparativement à l'expertise technique. Ainsi, nous pouvons distinguer parmi les différentes études plusieurs aptitudes non-techniques dont doit disposer l'architecte :

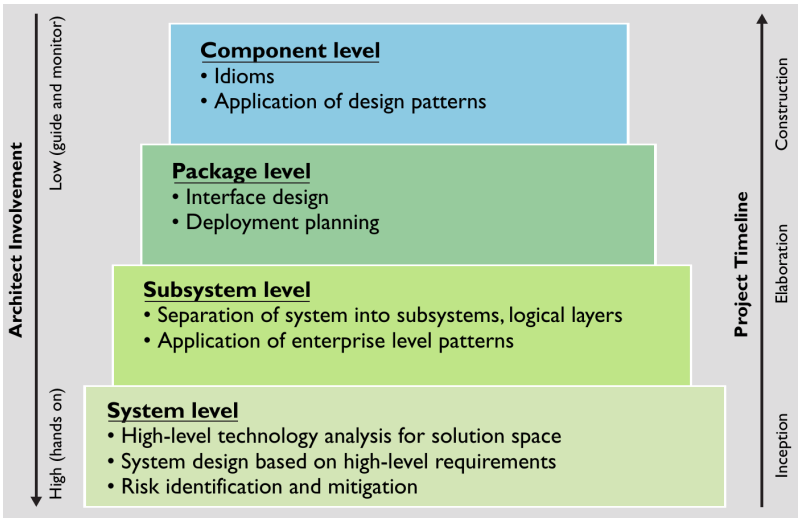



FIGURE I.2.2 – Niveaux du système que l'architecte doit gérer durant le cycle de vie [McB07].

- la communication. Celle-ci doit être claire et efficace. Du fait de son rôle, l'architecte est en lien avec de multiples parties prenantes (développeurs, clients, comité exécutif, etc.). Il doit ainsi savoir adapter son discours. Cette efficacité repose sur un discours borné, conceptualisé et synthétique. De plus l'architecte se doit d'être à l'écoute des différentes parties prenantes [Kru99, McB07].
- le *leadership*. L'architecte, au-delà de la gestion technique, est également un décisionnaire. Il est responsable de décisions techniques impactant fortement le développement mais aussi le cycle de vie du logiciel. Ainsi, il doit faire preuve d'assurance dans ses décisions et être moteur dans le projet. Cela passe notamment par l'insufflation de nouvelles idées, technologies et méthodes aux équipes de développement [Kru99]. Il doit également être capable de gérer, diriger et accompagner ses équipes [McB07, SH15].
- le *marketing* et le *business*. L'architecte est amené à rencontrer de nombreuses parties-prenantes dont certaines sont les représentantes des entités *business*, *marketing* ou de l'exécutif d'une entreprise. L'architecte doit pouvoir dialoguer avec ces entités dans un langage clair et compréhensible par les décisionnaires [McB07]. Il doit également connaître la valeur ajoutée de son produit par rapport à la concurrence [Kru99, McB07]. En outre, il doit être capable de chiffrer son projet et de projeter des estimations de coûts [SH15].

Les différents travaux décrits ici [Son98, Kru99, SNV06, McB07, Ber08, SH15] donnent une vision de l'architecte et montrent la pluralité des compétences dont il doit faire preuve. Parmi ces compétences certaines sont mesurables tandis que d'autres non. Les compétences techniques de l'architecte et son expérience dans le temps sont des notions mesurables en utilisant, par exemple, des métriques logicielles. Les compétences non-techniques, les *soft-skills*, ne sont, quant à elles, pas mesurables directement.

## 2.6 Apprentissage automatique pour le génie logiciel

Avec l'augmentation constante de la taille et de la complexité des logiciels, la production logicielle est soumise à des problèmes de gestion des coûts de développement et de la qualité sans cesse renouvelés. Le test logiciel devient plus important et doit satisfaire des exigences de qualité toujours plus hautes. La maintenance se complexifie également. La gestion même du projet devient centrale pour la réussite des développements. L'émergence de l'utilisation de l'apprentissage automatique en génie logiciel répond à ces besoins de productivité et d'automatisation.

### 2.6.1 Types d'apprentissage automatique

L'apprentissage automatique aussi appelé *machine learning* est un domaine de l'intelligence artificiel regroupant plusieurs types d'apprentissage :

- L'apprentissage supervisé consiste à apprendre la correspondance entre une entrée et une sortie sur la base d'exemples de paires entrée-sortie [RN20]. La sortie peut être une étiquette (classification) ou une valeur (régression). Plus formellement, nous pouvons définir l'apprentissage supervisé comme un système se basant sur un espace $X$ constitué de $n$ observations (lignes) et $p$ variables (colonnes) et un vecteur d'étiquettes $Y$ de taille $n$

$$X = \begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,p} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,p} \\ \vdots & \vdots & \cdots & \vdots \\ x_{n,1} & x_{n,2} & \cdots & x_{n,p} \end{pmatrix} \text{ et } Y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} \tag{I.2.1}$$

  Dans l'Équation (I.2.1), $x_{i,j}$ est une valeur de l'observation $i$ sur la variable $j$,
  $X_i = \{x_{1,1}, x_{1,2}, ..., x_{1,p}\}$ représente l'observation disponible à la ligne i. Sachant que chaque observation $X_i$ est associée à une étiquette $y_i$, on suppose que l'on peut associer les étiquettes de $Y$ aux observations de $X$ à l'aide d'une fonction $f : X \longmapsto Y$. Cependant, les données étant potentiellement bruitées ou incomplètes et la nature de la relation entre X et Y étant inconnue, la relation est approximative et non déterministe. De fait, le problème se modélise alors comme une fonction $f : X \longmapsto Y + \epsilon$ où $\epsilon$ est l'erreur à minimiser dans le but d'obtenir une association entre $X_i$ (observation $i$) et $y_i$ (étiquette $i$) la plus précise possible. La Figure I.2.3 illustre le mécanisme d'apprentissage supervisé.
  Les systèmes d'apprentissage supervisé permettent de résoudre deux types de problèmes de classification :

  - classification binaire : l'espace des étiquettes est binaire $Y = \{0, 1\}$.
  - classification multi-classes : l'espace des étiquettes est discret et fini de sorte que $Y = \{0, 1, ..., c\}$ où $c$ est le nombre de classes.

- L'apprentissage non-supervisé intervient lorsqu'on dispose d'un ensemble de données $X$ mais pas du vecteur d'étiquettes $Y$. Ce type d'apprentissage est constitué d'algorithmes capables

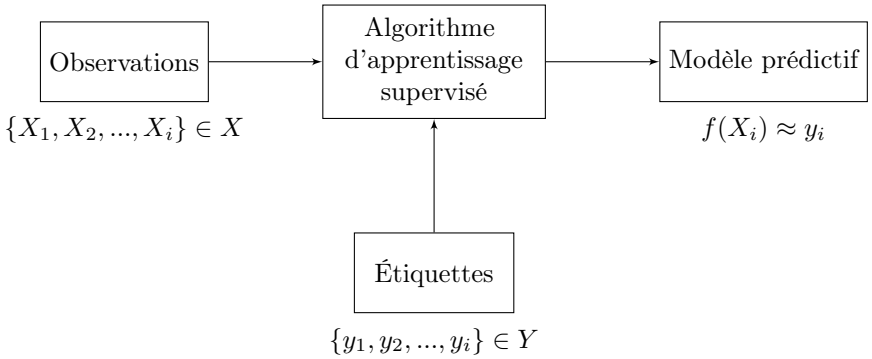



FIGURE I.2.3 – Mécanisme de l'apprentissage supervisé.

d'apprendre des régularités (*patterns*) parmi des données non-étiquetées. Ce type d'apprentissage est le plus souvent utilisé pour le partitionnement de données (*clustering*). Ce partitionnement peut se définir formellement comme la recherche de $k$ partitions $C_i$ parmi l'espace $X$ de telle sorte que $X = \bigcup_{i=1}^{k} C_i$. La Figure I.2.4 illustre le mécanisme d'apprentissage supervisé.

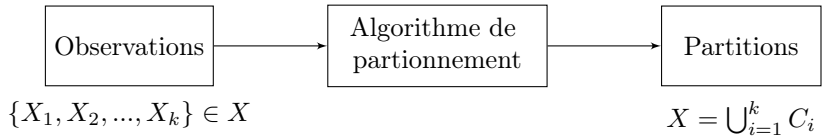



FIGURE I.2.4 – Mécanisme de partitionnement en apprentissage non-supervisé.

- L'apprentissage semi-supervisé : est une approche combinant des données étiquetées et non-étiquetées. Elle se situe à mi-chemin entre l'apprentissage supervisé et non-supervisé. Le principe est de tenir compte des données étiquetées pour résoudre des problématiques d'apprentissage supervisé ou non-supervisé. Ce type d'apprentissage permet par exemple d'étiqueter de grands jeux de données.
- L'apprentissage par renforcement : consiste pour un agent dans un environnement changeant ou non à apprendre des suites d'actions à entreprendre par expérience de manière à maximiser une récompense.
- L'apprentissage par transfert : fonctionne sur la base de connaissances apprises sur des tâches antérieures puis appliquées à de nouvelles tâches connexes au domaine d'apprentissage.

### 2.6.2 Mesures de performance utilisées en apprentissage automatique

Dans cette thèse, nous allons mesurer la performance des systèmes d'apprentissage automatique avec quatre mesures classiques tenant compte de la part d'éléments catégorisés comme vrais positifs ($VP$), faux positifs ($FP$), vrais négatifs ($VN$) et faux négatifs ($FN$) (voir Figure I.2.5).

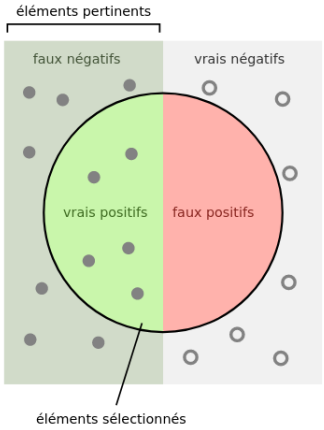



FIGURE I.2.5 – Visualisation des vrais positifs (VP), des faux positifs (FP), des vrais négatifs (VN) et des faux négatifs (FN).

La première mesure concerne la précision de la classification. Elle correspond au nombre de candidats pertinents sélectionnés parmi l'ensemble des candidats sélectionnés. La précision est définie par la formule suivante :

$$précision = \frac{VP}{(VP + FP)} \tag{I.2.2}$$

La seconde concerne le rappel. Le rappel mesure le nombre d'éléments pertinents sélectionnés parmi l'ensemble des candidats pertinents potentiels. Le rappel est ainsi défini par :

$$rappel = \frac{VP}{(VP + FN)} \tag{I.2.3}$$

La troisième mesure est la mesure F1. Cette mesure est une moyenne harmonique entre la précision le rappel. Elle détermine la capacité du système à donner toutes les solutions pertinentes et à refuser les autres. La mesure F1 est définie par la formule suivante :

$$F1 = 2 \cdot \frac{(précision \cdot rappel)}{(précision + rappel)} \tag{I.2.4}$$

Enfin nous mesurons la justesse. Elle mesure la part de prédictions correctes par le classifieur par rapport au nombre total d'observations. La justesse est définie par la formule suivante :

$$justesse = \frac{VP + VN}{VP + VN + FP + FN} \tag{I.2.5}$$

### 2.6.3 Domaines d'application de l'apprentissage automatique en génie logiciel

Le génie logiciel et l'apprentissage automatique peuvent se conjuguer dans la résolution de problèmes posés par le génie logiciel [Gir21]. L'apprentissage automatique pour le génie logiciel a émergé au début des années 90. De manière non-exhaustive nous pouvons citer les champs d'application suivants :

- la variabilité logicielle et les lignes de produits : générations de configurations [AMP+19], inférences de contraintes pour les lignes de produit [TGAJ16], etc.
- le test logiciel : génération de mutants [MZS+18], prédiction de défauts [Gon08], génération de tests sur la base de traces d'exécutions [ULD+20], etc.
- la gestion de tickets : classification [THPM17], attribution des tickets [JKZ09], etc.
- la qualité : détection de *code smells* [FMZM16].
- la gestion de projet : prédiction des coûts de développement [SF95].

## 2.7 Motivations

L'utilisation de l'apprentissage automatique permet de répondre aujourd'hui à de nombreuses problématiques en génie logiciel. Associé à des métriques et aux artefacts gravitant autour du code (tickets logiciels, commits, etc.), l'apprentissage automatique permet la création de systèmes d'assistance à la prise de décisions, permettant gain de temps et d'efficacité. Le Chapitre II.4 répond à un problème de traitement de masse des informations par la classification automatique de tickets de bogues.

L'architecture est un élément central dans la conception logicielle. De celle-ci découle le métier d'architecte et l'expertise nécessaire à la bonne conception et réalisation logicielle. Cette expertise comporte à la fois un champs potentiellement mesurable à travers des métriques mais aussi un aspect moins directement quantifiable lié aux *soft skills*. Ainsi, savoir mesurer la contribution des développeurs à l'architecture ouvre des perspectives pour la gestion de projet et le management des équipes de développement. Ainsi, le Chapitre II.5 répond à cette problématique en proposant un modèle de mesure de la contribution à l'architecture *runtime*.

Le Chapitre II.6 montre une méthode de classification des développeurs expérimentés pouvant être utilisée en conduite de projet.

Dans le Chapitre II.7, nous croisons les résultats obtenus au Chapitre II.5 avec la méthode de classification du Chapitre II.6 afin d'étudier le profil des développeurs expérimentés du point de vue de leurs contributions aux architectures *runtime*.

## 2.8 Conclusion

Ce chapitre expose le contexte général de cette thèse en introduisant les concepts liés aux architectures à l'exécution. Nous avons ensuite présenté l'expertise des développeurs et celle rattachée plus spécifiquement aux architectes. Nous avons enfin abordé les différents types de métriques logicielles ainsi que l'application de l'apprentissage automatique au domaine du génie logiciel.
Ce chapitre identifie les enjeux liés au développement des architectures et à l'identification des personnes responsables de celles-ci sur la base de métriques spécifiques. Un autre enjeu présenté ici est lié à l'utilisation de l'apprentissage automatique sur des problématiques de classification. Nous appliquons dans cette thèse la classification automatique pour réaliser l'identification de développeurs expérimentés sur la base de métriques logicielles. Un second problème adressé concerne le traitement de grande masses d'informations en lien avec le logiciel. Nous étudions dans cette thèse le problème de classification automatique de tickets logiciels et plus particulièrement de ceux décrivant des bogues. L'ensemble de ces problèmes s'inscrivent dans le cadre d'un objectif global d'assistance à la prise de décisions et à la gestion de projets.

# Chapitre I.3

# État de l'art

## Sommaire

Dans ce chapitre, nous exposons les travaux sur lesquels nous appuyons nos recherches ainsi que les travaux connexes, sources d'inspiration ou de comparaison.
Une première partie de ce chapitre est dédiée à l'étude des jeux de données de tickets logiciels. Nous détaillons ensuite les approches utilisant ces jeux de données pour la classification de tickets. Une deuxième partie de cet état de l'art porte sur les travaux étudiant la contribution aux développement de logiciels.
Enfin, la dernière partie de ce chapitre est consacrée aux approches mesurant l'expertise des développeurs dans les projets logiciels.

## 3.1 Travaux sur la classification automatique de tickets

### 3.1.1 Jeux de données existants

L'apprentissage supervisé requiert un jeu de données étiquetées afin d'entraîner un classifieur. Ce jeu de données sert à la fois pour l'entraînement du classifieur et pour le test des performances de la classification. Diverses études ont contribué à créer des jeux de données contenant des tickets issus de multiples projets logiciels. Nous faisons ici état des jeux de données utilisés par les approches décrites dans la Section 3.1.2. La colonne "jeu de données utilisé" de la Table I.3.3 récapitule les différents jeux que nous présentons ici.

#### 3.1.1.1 Antoniol *et al.*

Antoniol *et al.* [AAP+08] ont produit en 2008 un jeu de données de 1800 tickets issus de trois projets *open-source* : Eclipse, Mozilla et JBoss. Le détail du jeu d'Antoniol *et al.* est donné par la Table I.3.1. Ce jeu de données a été nettoyé manuellement afin d'éviter de mauvais étiquetages par les rédacteurs des tickets. Le jeu données d'Antoniol *et al.* n'est pas mis à disposition en ligne.

| Projet | #Bogue | #Non-Bogue | #Total |
|---|---|---|---|
| Mozilla | 270 | 330 | 600 |
| Eclipse | 194 | 406 | 600 |
| JBoss | 345 | 255 | 600 |

TABLE I.3.1 – Détails du jeu de données de Antoniol *et al.* [AAP+08].

#### 3.1.1.2 Herzig *et al.*

Herzig *et al.* [HJZ13] ont produit un jeu de données pour leur étude sur la mauvaise classification des tickets par les développeurs. Herzig *et al.* ont montré qu'environ 30% des tickets étaient mal étiquetés. Le jeu de données est constitué de 5591 tickets issus de l'ITS Jira [1] et annotés manuellement par Herzig *et al.*. Ces tickets proviennent de trois projets *open-source* : HTTPClient,

1. Jira (https://www.atlassian.com/fr/software/jira) est un des *Issue Tracking Systems* (ITS) les plus utilisés dans les projets logiciels.

Lucene et JackRabbit. La Table I.3.2 détaille la composition du jeu de données. Un avantage majeur de ce jeu de données est sa mise à disposition en ligne par les auteurs [2]. Une conséquence induite est la sa forte réutilisation par d'autres travaux de recherche faisant des classifications [PHM13, CS15, PSHS17, THPM17, QS18, LSK+19].

| Projet | #Bogue | #Non-Bogue | #Total |
|---|---|---|---|
| HTTPClient | 305 | 441 | 746 |
| Jackrabbit | 697 | 1746 | 2443 |
| Lucene-Java | 938 | 1464 | 2402 |

TABLE I.3.2 – Détails du jeu de données de Herzig *et al.* [HJZ13].

#### 3.1.1.3 Otoom *et al.*

Otoom *et al.* [OAH19] ont proposé un jeu de données différent des travaux précédents [AAP+08, HJZ13]. En effet, leur jeu de données ne contient que des tickets de bogues décrivant soit des bogues qualifiés de correctifs, qui correspondent à la description de défauts dans le logiciels, soit des bogues qualifiés de perfectifs, qui eux correspondent à des demandes de maintenance (mise à jour de bibliothèques, amélioration de performances, etc.) du logiciel. Le jeu de données de Otoom *et al.* contient au total 5800 tickets issus de trois projets logiciels *open-source*: Tomcat (1056), AspectJ (593) et SWT (4151). Ce jeu n'a pas été mis à disposition en ligne par les auteurs.

#### 3.1.1.4 Kallis *et al.*

Kallis *et al.* [KSCP19] ont bâti un jeu de données destiné à des expérimentations de classification de tickets. Le jeu de données de Kallis *et al.* contient 34000 tickets issus de 12112 projets *open-source*. Il propose trois catégories de tickets : bogues (16355), questions (14228) et améliorations (3458). Ce jeu de données n'a pas été annoté manuellement et est donc soumis aux risques de mauvaise classification décrits par Herzig *et al.* [HJZ13]. Le jeu de données est mis à disposition en ligne [3] par Kallis *et al.*.

### 3.1.2 Approches de classification de tickets

La gestion des bogues a toujours été une préoccupation majeure en qualité logicielle. La création des systèmes de gestion de tickets a permis aux utilisateurs et développeurs de décrire les problèmes auxquels ils sont confrontés. L'émergence de ces systèmes et leur démocratisation permet la mise en place d'approches pour le traitement automatique des tickets. Ici, nous abordons la classification automatique de tickets. Plusieurs travaux abordent cette problématique avec des approches et des jeux de données différents.

2. https://github.com/qperez/ATTIC/tree/main/data
3. https://tinyurl.com/y23kgdro

| Étude | Date | Type de classification | | Type(s) de classifieur(s) utilisé(s) | Catégories de tickets traitées | Jeu de données utilisé | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Binaire | Multi-classes | | | Étude initiale | #Projets | #Tickets | Annoté manuellement | Mis en ligne |
| Antoniol *et al.* [AAP+08] | 2008 | ✔ | ✖ | *Decision Tree*<br>*Naive Bayes*<br>*Logistic Regression* | *Bogue*<br>*Non-bogue* | Antoniol *et al.* [AAP+08] | 3 (Eclipse Mozilla jBoss) | 1800 | ✔ | ✖ |
| Pingclasai *et al.* [PHM13] | 2013 | ✔ | ✖ | *Decision Tree*<br>*Naive Bayes*<br>*Logistic Regression* | Bogue<br>Non-bogue | Herzig *et al.* [HJZ13] | 3 (HTTPClient Lucene JackRabbit) | 5591 | ✔ | ✔ |
| Limsettho *et al.* [LHM14] | 2014 | ✔ | ✖ | *Decision Tree*<br>*Naive Bayes*<br>*Logistic Regression*<br>*Voting Classifier* | Bogue<br>Non-bogue | Herzig *et al.* [HJZ13] | 3 (HTTPClient Lucene JackRabbit) | 5591 | ✔ | ✔ |
| Chawla *et al.* [CS15] | 2015 | ✔ | ✖ | Régles de Logique floue | Bogue<br>Non-bogue | Herzig *et al.* [HJZ13] | 3 (HTTPClient Lucene JackRabbit) | 5591 | ✔ | ✔ |
| Terdchanakul *et al.* [THPM17] | 2017 | ✔ | ✖ | *Logistic Regression*<br>*Random Forest* | Bogue<br>Non-bogue | Herzig *et al.* [HJZ13] | 3 (HTTPClient Lucene JackRabbit) | 5591 | ✔ | ✔ |
| Pandey *et al.* [PSHS17] | 2017 | ✔ | ✖ | *Naive Bayes*<br>*Linear Discriminant Analysis*<br>*k-Neareast Neighbors*<br>*Support Vector Machine*<br>*Decision Tree*<br>*Random Forest* | Bogue<br>Non-bogue | Herzig *et al.* [HJZ13] | 3 (HTTPClient Lucene JackRabbit) | 5591 | ✔ | ✔ |
| Qin et Sun *et al.* [QS18] | 2018 | ✔ | ✖ | *Long Short-Term Memory* avec couche SoftMax | Bogue<br>Non-bogue | Herzig *et al.* [HJZ13] | 3 (HTTPClient Lucene JackRabbit) | 5591 | ✔ | ✔ |
| Otoom *et al.* [OAH19] | 2019 | ✔ | ✖ | *Support Vector Machine* | Bogues correctifs (défaut)<br>Bogues perfectifs (maintenance) | Otoom *et al.* [OAH19] | 3 (AspectJ Tomcat SWT) | 5800 | ✔ | ✖ |
| Luaphol *et al.* [LSK+19] | 2019 | ✔ | ✖ | *Logistic Regression*<br>*Naive Bayes*<br>*Support Vector Machine* | Bogue<br>Non-bogue | Herzig *et al.* [HJZ13] | 3 (HTTPClient Lucene JackRabbit) | 5591 | ✔ | ✔ |
| Kallis *et al.* [KSCP19] | 2019 | ✖ | ✔ | Modèle FastText avec couche SoftMax | Bogue<br>Question<br>Amélioration | Kallis *et al.* [KSCP19] | 12112 | 34000 | ✖ | ✔ |

TABLE I.3.3 – Approches existantes dédiées à la classifications de tickets.

| Étude | Mesure F1 | | | | | Protocole d'évaluation |
|---|---|---|---|---|---|---|
| | HTTPClient | Jackrabbit | Lucence | Moyenne Projets | Cross-Project | |
| Terdchanakul *et al.* [THPM17] | 0.814 | 0.805 | 0.884 | 0.834 | 0.814 | 10-fold |
| Chawla *et al.* [CS15] | 0.830 | 0.780 | 0.840 | 0.817 | – | Training/Test split 80/20 |
| Pingclasai *et al.* [PHM13] | 0.758 | 0.767 | 0.818 | 0.781 | – | 10-fold |
| Luaphol *et al.* [LSK+19] | – | – | – | – | 0.770 | Training/Test split 70/30 |
| Pandey *et al.* [PSHS17] | 0.687 | 0.759 | 0.708 | 0.718 | – | 10-fold |
| Qin et Sun [QS18] | 0.757 | 0.771 | 0.717 | 0.748 | 0.746 | Training/Test split 90/10 |

TABLE I.3.4 – État de l'art sur la classification de tickets avec le jeu de données de Herzig *et al.* [HJZ13].

### 3.1.3 Approches de classification binaire de tickets de bogues utilisant le jeu de données de Herzig *et al.*

Comme montré par la Table I.3.3, sept approches utilisent le jeu de données de Herzig *et al.* [HJZ13]. Nous allons dans cette section détailler les approches mises en oeuvre ainsi que les résultats obtenus. La Table I.3.4 résume les performances obtenues par les différentes travaux ainsi que le protocole d'évaluation utilisé pour réaliser la mesure. Dans la Table I.3.4, les intervalles de confiance ne sont pas indiqués car non fournis pas les auteurs.

#### 3.1.3.1 Qin et Sun

Qin et Sun [QS18] ont proposé une approche basée sur un réseau de neurones du type *Long Short-Term Memory* couplé à une couche de neurones de sortie utilisant une fonction exponentielle normalisée (couche SoftMax) afin de classifier des tickets de bogues. La vectorisation du corpus est effectuée par plongement lexical (*word embedding*). Cette technique permet d'associer aux différents mots des tickets les contextes dans lesquels ils sont employés. Les résultats obtenus par Qin et Sun sont mitigés, avec une mesure F1 de 0.746 sur l'ensemble du corpus comparativement à d'autres approches utilisant des classifieurs plus simples comme la régression logistique ou les arbres de décisions. De plus, Quin et Sun évaluent leur modèle en divisant le jeu de données en deux parties (*train/test split*) : 90% du jeu sont utilisés pour entraîner le modèle tandis que 10% seulement sont utilisés pour l'évaluer. Cependant, cette méthode d'évaluation est moins robuste qu'une évaluation *k-fold*.

#### 3.1.3.2 Pandey *et al.*

Pandey *et al.* [PSHS17] ont comparé plusieurs types de classifieurs (*Random Forest*, *Decision Tree*, *Support Vector Machine*, *k-Neareast Neighbors*, *Linear Discriminant Analysis*, *Naive Bayes*) afin de sélectionner le meilleur. Les résultats présentés par la Table I.3.4 ont été obtenus grâce à un

classifieur de type *Random Forest*. Les données textuelles du corpus de tickets sont utilisées afin de créer une matrice terme-document où chaque ligne de la matrice représente un document (ticket) et chaque colonne un terme présent dans le corpus de tickets. Ainsi, chaque valeur de la matrice représente le nombre d'occurrences d'un terme dans un ticket. Les résultats obtenus par Pandey *et al.* sont plus faibles que ceux obtenus par Qin et Sun [QS18]. En revanche, l'évaluation de leur approche de classification a été faite à l'aide d'un $10-fold$ assurant une plus grande robustesse à leurs résultats.

#### 3.1.3.3 Luaphol *et al.*

Luaphol *et al.* [LSK+19] ont testé trois classifieurs (*Logistic Regression*, *Naive Bayes*, *Support Vector Machine*) sur le jeu de données de Herzig *et al.* [HJZ13]. À contrario des autres travaux, Luaphol *et al.* ne donnent qu'un résultat global sur l'ensemble du corpus de tickets (voir Table I.3.4). Luaphol *et al.* vectorisent les données textuelles à l'aide de la méthode du sac de mots (*bag of words*) où chaque document est représenté comme le "sac" des mots qu'il contient sans tenir compte de la grammaire ou de l'ordre des mots. Le sac de mots permet de calculer la fréquence des termes dans les différents documents. Cette fréquence est pondérée par les auteurs en utilisant la méthode *term frequency–inverse document frequency* (TF-IDF). Cette méthode permet de relativiser l'importance d'un terme en tenant compte de sa fréquence dans l'entièreté du corpus (un terme fréquent dans tous les documents est moins discriminant). En couplant ce traitement textuel avec un classifieur du type *Logistic Regression*, les auteurs obtiennent 0.770 pour la mesure F1 sur l'ensemble du corpus. Cependant, ce score est à relativiser car Luaphol *et al.* ne donnent pas les résultats pour chacun mais seulement la moyenne sur le corpus complet (cross-project). De plus, la méthode d'évaluation utilisée (*train/test split*) ne permet pas une évaluation robuste des résultats comme avec la méthode *k-fold*.

#### 3.1.3.4 Pingclasai *et al.*

Pingclasai *et al.* [PHM13] mettent en œuvre une approche basée sur de la modélisation de sujets (*topic modeling*) permettant d'entraîner trois types de classifieurs (*Decision Tree*, *Naive Bayes*, *Logistic Regression*) afin d'en comparer les résultats. La modélisation de sujets permet d'abstraire par des moyens probabilistes les termes associés à un ou plusieurs sujets thématiques. En utilisant cette approche, les auteurs obtiennent des performances relativement bonnes grâce aux différents classifieurs testés : 0.758 sur le projet HTTPClient en utilisant un *Decision Tree*, 0.767 sur le projet Jackrabbit et 0.818 sur le projet Lucene avec une *Logistic Regression*. Néanmoins, ces résultats sont à nuancer car obtenus avec une évaluation divisant le jeu de données en deux (*train/test split*) et la mesure F1 sur l'ensemble du corpus de tickets n'est pas donnée.

#### 3.1.3.5 Chawla *et al.*

Chawla *et al.* [CS15] utilisent des règles de logique floue afin de classifier les tickets. Les règles sont créés en fonction de l'appartenance d'un terme à la catégorie des tickets référencés comme des bogues. Chaque règle retourne une valeur d'appartenance d'un terme issu d'un ticket dans l'intervalle $[0, 1]$. Ainsi, en construisant le produit des règles pour chaque terme d'un ticket, il est possible de donner une probabilité d'appartenance à la catégorie bogue. L'approche de Chawla *et al.* donne de bons résultats : 0.830 sur le projet HTTPClient, 0.780 sur le projet Jackrabbit et 0.840 sur le projet Lucene. Chawla *et al.* obtiennent de bons résultats sur la moyenne des trois projets étudiés (0.817). En revanche, ils ne donnent pas de valeur pour la mesure F1 inter-projets, ce qui ne permet pas de connaître la performance sur l'ensemble du corpus. De plus, Chawla *et al.* n'utilisent pas un protocole d'évaluation robuste comme *k-fold* mais un *train/test split*.

#### 3.1.3.6 Terdchanakul *et al.*

Terdchanakul *et al.* [THPM17] obtiennent les meilleurs résultats de classification à la fois sur l'ensemble du jeu de données (0.814) mais également sur chacun des projets séparément (0.805 pour Jackrabbit et 0.884 pour Lucene) à l'exception de HTTPClient où la mesure F1 (0.814) est légèrement en dessous des résultats de Chawla *et al.* [CS15] (0.830). L'approche mise en œuvre par les auteurs utilise la méthode N-gramme IDF [SHN15] pour transformer les informations textuelles en vecteurs. Sur la base de ces vecteurs, un sous-ensemble des plus représentatifs du jeu de tickets est effectué à l'aide de la méthode du khi-deux. Ce sous-ensemble de vecteurs représentatifs est ensuite utilisé pour l'entraînement de classifieurs du type *random forest* et *logistic regression*. Les performances obtenues par l'approche de Terdchanakul *et al.* [THPM17] sont notables. La performance sur HTTPClient (0.814) est obtenue à l'aide d'un classifieur de type *random forest*.

### 3.1.4 Approches liées aux autres jeu de données

Les auteurs des approches présentées ici sont également ceux ayant créé les jeux de données présentés en Section 3.1.1. Sur chacun de ces jeux, nous ne disposons que d'une seule approche. Il nous est alors difficile de procéder à une analyse comparative des résultats obtenus.

#### 3.1.4.1 Antoniol *et al.*

Antoniol *et al.* [AAP+08] sont les premiers à avoir créé un jeu de données de tickets et à avoir construit une approche autour de celui-ci. Les auteurs entraînent et comparent trois types de classifieurs : *decision tree*, *naive bayes* et *logistic regression* sur leur jeu de données de 1800 tickets. Les mots des tickets sont traités afin d'en extraire la racine (*stemming*) puis les fréquences des racines extraites sont calculées. Ces fréquences constituent des vecteurs qui sont utilisées pour entraîner les classifieurs. La validation de l'approche est faite à l'aide d'un 10-$fold$ permettant des résultats robustes. Antoniol *et al.* obtiennent de bons résultats de précision (0.82) et de rappel (0.83) à l'aide d'un classifieur de type *logistic regression*. De plus, les auteurs ont montré que leur

| | Bogues | Amélioration | Questions |
|---|---|---|---|
| F-Mesure | 0.822 | 0.894 | 0.781 |
| Précision | 0.841 | 0.763 | 0.874 |
| Rappel | 0.831 | 0.823 | 0.825 |

TABLE I.3.5 – Résultats obtenus par Kallis *et al.* [KDCP21].

approche est plus efficace que l'utilisation d'expressions régulières pour la détection de ticket de bogues.

#### 3.1.4.2 Kallis *et al.*

Kallis *et al.* [KDCP21] proposent un classifieur multi-classes permettant de catégoriser les tickets en trois grandes familles : bogues, questions et améliorations. Ils utilisent comme classifieurs des réseaux de neurones créés et pré-entraînés par Facebook, le modèle FastText [4]. L'entraînement complémentaire est ensuite réalisé selon le principe du *transfert learning* sur un corpus de 30000 tickets provenant de 12112 projets hébergés sur GitHub. Kallis *et al.* n'ont pas nettoyé manuellement le corpus de tickets et les étiquettes utilisées sont celles données par les rédacteurs des tickets. En procédant de cette manière, il est probable que certains tickets aient un mauvais étiquetage et que la qualité du modèle final en souffre. La transformation des données textuelles en vecteurs utilisables par le classifieur est réalisée avec la méthode du sac de mots. Comme indiqué dans la Table I.3.5, l'utilisation de FastText par Kallis *et al.* permet d'obtenir de bonnes performances.

### 3.1.5 Discussion

Plusieurs jeux de données existent pour la classification de tickets. Parmi ces jeux, certains ont été utilisé de manière unique [AAP+08, KSCP19, OAH19] tandis que celui de Herzig *et al.* [HJZ13] a été repris dans de nombreuses approches [PHM13, CS15, PSHS17, THPM17, QS18, PSHS17]. Cela vient notamment du fait qu'Herzig *et al.* ont mis en ligne leur jeu de données mais également de la consolidation manuellement réalisée sur celui-ci.
Parmi les approches utilisant le jeu de données de Herzig *et al.*, trois études [PHM13, CS15, PSHS17] ne fournissent pas la mesure F1 inter-projets. La seule manière de comparer approximativement leurs résultats sur la globalité des tickets du jeu de données est alors de calculer la valeur moyenne des résultats obtenus sur les différents projets. Les protocoles d'évaluation sont également différents selon les études.
Seules trois études [THPM17, PHM13, PSHS17] utilisent un protocole d'évaluation robuste, à savoir la validation croisée $k$-fold ($k$=10). Les autres travaux [CS15, PSHS17, LSK+19] utilisent un protocole moins robuste (*train*/*test split* 90/10 ou 70/30). La mesure croisée la plus faible est obtenue par Qin *et al.* [QS18] (0,746) en utilisant une répartition *entraînement*/*test* 90/10. Les meilleurs résultats sont ceux de Terdchanakul *et al.* [THPM17] sur toutes les mesures. Ces résultats

4. https://fasttext.cc/

| Étude | Type de mesure | Seuil pour la mesure | Formalisation de la mesure | Granularité | Données utilisées |
|---|---|---|---|---|---|
| Mockus *et al.* [MFH02] | Nombre de changements effectués par les développeurs | ✖ | ✖ | - Fichier | Données CVS d'Apache HTTP Server et de Mozilla Firefox |
| Bird *et al.* [BNM+11] | Mesure du taux de propriété | ✔ (défini à 5%) | ✖ | - dll | Historique des projets Windows 7 et Vista |
| Foucault *et al.* [FFB14] | Mesure du taux de propriété | ✔ (défini à 5%) | ✔ | - Fichier<br>- Package | Données des systèmes de gestion de versions des 7 projets open sources étudiés |

TABLE I.3.6 – Travaux autour de la mesure de la contribution des développeurs.

ont été obtenus en utilisant la méthode N-gramme IDF et ainsi que des classifieurs *random forest* et *logisitic regression*. Notons que les auteurs utilisent un protocole d'évaluation robuste, ce qui en fait la référence choisie pour comparer nos résultats dans la suite de cette thèse.
Les approches sur les autres jeux de données utilisent différentes méthodes de classification comme par exemple le modèle pré-entraîné FastText qui permet à Kallis *et al.* [KSCP19] d'obtenir de bons résultats sur leur jeu de données. Otoom *et al.* [OAH19] se sont focalisés sur des sous-types de tickets de bogues qu'ils ont classés à l'aide d'un classifieur de type *Support Vector Machine*.
Les performances obtenues pas les différents types de classifieurs sont très variables dans l'état de l'art et ne font pas ressortir un type de classifieur à privilégier pour réaliser la classification de tickets ou de critères permettant de choisir quel type de classifieur sélectionner en fonction des spécificités de chaque approche de classification ou de chaque jeu de données. Aussi, nous conduisons dans cette thèse une évaluation des performances de différents types de classifieurs, afin de déterminer le plus adapté à notre approche.

## 3.2 Mesure de la contribution logicielle

Le mesure de la contribution logicielle a pu voir le jour grâce à la création et à la démocratisation des systèmes de gestion de versions comme CVS (1990), Subversion (2000) ou encore Git (2005). Cette métrique se mesure en utilisant les changements enregistrés par les systèmes de gestion de versions afin de distinguer par développeurs un taux de propriété du code (*code ownership*). Les métriques de mesure de la contribution logicielle font partie intégrante de la catégorie des métriques de process décrites en Section 2.3.3. Nous allons ici présenter les travaux liées à la création de métriques de mesure des taux de contribution. La Table I.3.6 présente les différentes approches que nous allons décrire.

### 3.2.1 Travaux de Mockus *et al.*

En 2002, Mockus *et al.* [MFH02] ont été les premiers à mesurer la contribution logicielle des développeurs sur deux projets : Apache HTTP Server et Mozilla Firefox. La mesure de la contribution

(*code ownership*) est faite par dénombrement du nombre de changements effectués par chaque développeur sur chaque fichier des projets étudiés. Mockus *et al.* ont choisi ces deux projets du fait de leurs spécificités : l'un (Apache HTTP server) est *open-source* sans visée commerciale tandis que l'autre (Mozilla Firefox) est *open-source* avec une visée commerciale. Mockus *et al.* ont montré dans le cas d'Apache qu'un groupe restreint de développeurs étaient à l'origine de plus de 10% des modifications dans les fichiers ayant fait l'objet de plus de 30 changements. Dans le projet Mozilla Firefox, du fait de son caractère plus "professionnel", les changements du code sont soumis à une révision par les pairs. Mockus *et al.* n'ont pas mis en évidence de patron de contribution spécifique. Les travaux de Mockus *et al.* préfigurent les travaux de Bird *et al.* [BNM+11] sur le groupement des développeurs en fonction de leurs taux de propriété.

### 3.2.2 Travaux de Bird *et al.*

En 2011, Bird *et al.* [BNM+11] ont étudié les contributions des développeurs sur des modules (librairies dll) de Windows Vista et Windows 7. Bird *et al.* ont défini le taux de propriété sur un module comme le ratio entre les contributions (nombre de *commits*) d'un développeur donné et la somme des contributions des autres développeurs. Bird *et al.* introduisent également le concept de développeurs mineurs et développeurs majeurs. Les développeurs mineurs ont un taux de propriété inférieur à 5% tandis que les développeurs majeurs ont un taux de propriété supérieur ou égal à 5%. Dans leur étude, Bird *et al.* montrent la pertinence statistique des seuils de catégorisation des développeurs à l'aide d'un test de sensibilité. Les auteurs ont ensuite montré qu'il existait dans les modules Windows étudiés un ou plusieurs contributeurs dits "héros" avec de très hauts taux de contribution. Bird *et al.* ont également montré que les composants ayant des contributeurs avec de hauts taux de contribution ont généralement moins de défauts que ceux ayant un nombreux de contributeurs mineurs importants. De plus les auteurs obtiennent de forts taux de corrélations entre le nombre de défauts et les taux de propriété.

### 3.2.3 Travaux de Foucault *et al.*

Foucault *et al.* [FFB14] ont répliqué l'étude de Bird *et al.* en utilisant un corpus de 7 projets Java *open-source*. Ne travaillant pas sur des modules Windows (librairies dll) comme Bird *et al.*, les auteurs ont choisi de conduire leur étude sur deux niveaux de granularité : une première concerne le taux de contribution sur les classes Java (*file granularity*), la seconde sur les *packages* Java (*package granularity*). Dans le cadre de leurs travaux, Foucault *et al.* ont également donné une définition formelle de la mesure de la contribution logicielle. Sur un corpus de projets *open-source*, Foucault *et al.* font des constatations différentes de Bird *et al.*. En effet, Foucault *et al.* montrent que la corrélation entre taux de propriété du code et défauts est seulement dans certains cas très légèrement supérieure à celle des métriques liée au code (complexité et taille). Les auteurs mettent également en avant le fait qu'il n'y a pas de distinction en termes de nombre de défauts entre un module ayant un ou plusieurs contributeurs "héro" et un module ayant une multiplicité de

contributeurs mineurs.

### 3.2.4 Discussion

Mockus *et al.* [MFH02] ont été les premiers à faire une étude sur la contribution des développeurs et à évaluer la répartition de celles-ci dans deux projets logiciels *open-source*. Pour cela, Mockus *et al.* ont utilisé le nombre de changements faits par les développeurs. Ces mesures, bien que basiques, ont permis une première approche de la mesure de la contribution. Suite à ces travaux, Bird *et al.* ont eux aussi mesuré la contribution logicielle par l'utilisation d'un ratio de propriété du code sur des librairies dll Windows. Le nombre de contributions de chaque développeur est divisé par le nombre total des contributions sur chaque librairie dll, ce qui permet de mesurer un taux de propriété du code. Bird *et al.* ont également défini un seuil permettant de catégoriser les développeurs en fonction de leurs taux de propriété (contributeurs mineurs ou majeurs). Foucault *et al.* ont reproduit l'expérimentation de Bird *et al.* dans un contexte *open-source*. Foucault *et al.* [FFB14] ont formalisé la mesure du taux de propriété du code et ont réutilisé le seuil proposé par Bird *et al.*. Leurs résultats, bien que différents, ont néanmoins permis de valider la pertinence d'une mesure du taux de contributions mais également le seuil catégorisant les développeurs. Aucun de ces travaux n'est focalisé sur un type particulier de contributions. Nous étudions dans cette thèse la possibilité de mesurer spécifiquement les contributions aux architectures *runtime* afin de catégoriser le niveau de responsabilité et d'implication des développeurs dans le développement et la maintenance des architectures.

## 3.3 Mesure de l'expertise des développeurs

Nous avons vu au Chapitre I.2 la pluralité des rôles, des compétences et des connaissances que peuvent avoir les développeurs et plus particulièrement les architectes logiciels. Ainsi, nous distinguons deux catégories d'approches permettant de détecter des experts et/ou catégoriser les développeurs dans des projets logiciels :

- les approches par regroupement (*clustering*) permettent de regrouper et catégoriser les développeurs en fonction de leurs compétences et/ou de leurs expériences sur un projet donné
- les approches par profilage (*profiling*) permettent de découvrir des experts d'une ou plusieurs technologies sur un projet.

Notons que ces deux catégories d'approches ne sont pas incompatibles et qu'il est envisageable de profiler puis de regrouper des développeurs. Dans cette section, nous allons détailler les différents travaux permettant le regroupement et le profilage de développeurs. La Table I.3.7 présente les approches que nous allons détailler par la suite.

| Approche | Date | Profilage | Regroupement | Profilage des compétences architecturales | Basée sur le code | Entrées | Sortie(s) |
|---|---|---|---|---|---|---|---|
| Santos *et al.* [SdASOF18] | 2018 | ✔ | ✖ | ✔ | ✖ | ◦ Méta-données GitHub<br>◦ Données Git<br>◦ Code source | Nuage de mots représentant l'expertise du développeur |
| CVExplorer [GF16] | 2016 | ✔ | ✖ | ✔ | ✖ | ◦ Méta-données GitHub<br>◦ Fichiers ReadMe | Nuage de mots représentant l'expertise du développeur |
| Hauff *et al.* [HG15] | 2015 | ✔ | ✖ | ✖ | ✖ | ◦ Méta-données GitHub<br>◦ Fichiers ReadMe<br>◦ Offres d'emploi | Graphe ontologique de concepts représentant l'expertise du développeur |
| XTic [TPF+14] | 2014 | ✔ | ✔ | ✔ | ✔ | ◦ Données des dépôts<br>◦ Code source | Compétences associées à des niveaux d'expérience |
| LIBTIC [TFMB13] | 2013 | ✔ | ✖ | ✖ | ✔ | ◦ Données Git<br>◦ Code source<br>◦ Documentation<br>◦ Bibliothèques Maven utilisées (jar) | Liste d'experts pour chaque bibliothèque utilisée |
| Di Bella *et al.* [DBSS13] | 2013 | ✖ | ✔ | ✖ | ✖ | ◦ Données Git<br>◦ Métriques liées au code | Quatre groupes de développeurs classés en fonction de leurs métriques. |
| Kagdi *et al.* [KHM08] | 2008 | ✖ | ✔ | ✖ | ✔ | ◦ Données de Subversion<br>◦ Code source | Un groupe d'expert pour chaque granularité du projet (fichier, *package*, système) |
| Schuler *et al.* [SZ08] | 2008 | ✔ | ✖ | ✖ | ✔ | ◦ Données CVS<br>◦ Code source | Liste de développeurs experts |
| Sindhgatta [Sin08] | 2008 | ✔ | ✖ | ✖ | ✔ | ◦ Données des dépôts<br>◦ Code source | Niveaux d'expertise par développeur pour chacune des technologies du projet |
| Expertise Browser [MH02] | 200-2 | ✔ | ✖ | ✖ | ✔ | ◦ Données des dépôts<br>◦ Code source<br>◦ Documentation | Domaines d'expertise associés à chacun des *artefacts* du projet (code, documentation, tests...) |

Table I.3.7 – Approches existantes dédiées au profilage et/ou au regroupement de développeurs.

### 3.3.1 Approches par regroupement de développeurs

L'approche par regroupement de développeurs s'inspire du concept de communauté de développeurs étudié par Nakakoji *et al.* [NYN+02]. Ce concept est représenté par la Figure I.3.1 et montre qu'il est possible de structurer une communauté en plusieurs couches successives autour d'un chef de projet.

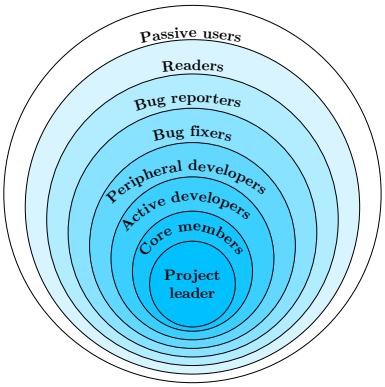



FIGURE I.3.1 – Concept de communauté autour d'un projet logiciel *open-source* de Nakakoji *et al.* [NYN+02].

Cette communauté va des utilisateurs qui sont les moins actifs dans le projet aux personnes rapportant ou fixant des bogues jusqu'aux membres les plus actifs et au chef de projet.

#### 3.3.1.1 Kagdi *et al.*

Kadgi *et al.* [KHM08] ont développé une approche permettant d'identifier trois groupes d'experts. Ces groupes correspondent à trois niveaux de granularité :

- fichier : développeurs ayant une expertise sur un fichier de code spécifique,
- *package* : développeurs ayant une expertise sur l'ensemble du code d'un *package*,
- système : développeurs ayant une expertise sur l'ensemble du projet.

Kadgi *et al.* mesurent l'expérience par un vecteur basé sur l'activité du développeur (nombre de *commits*) et le temps de travail passé par fichier. La somme de ces vecteurs sur l'ensemble des fichiers pour chaque développeur permet de mesurer leur expérience sur trois niveaux. De cette manière, Kadgi *et al.* sont en mesure de grouper les développeurs par niveaux. Ils ont évalué leur approche sur un panel de huit projets *open-source*. Cependant, avec cette approche, il n'est pas possible de profiler ou de grouper spécifiquement des développeurs ayant une expertise architecturale.

#### 3.3.1.2 Di Bella *et al.*

L'approche de Di Bella *et al.* [DBSS13] réutilise le concept de Nakakoji *et al.* [NYN+02] pour regrouper les développeurs en quatre catégories hiérarchiques (Figure I.3.2) :

| Acronyme de la variable | Variable | Description |
|---|---|---|
| DiP | *Days in the Project* | Nombre de jours passés dans le projet pour un développeur donné (temps entre le premier et le dernier *commit*) |
| NoC | *Number of Commits* | Nombre de modifications atomiques faites sur les fichiers de code source par un développeur donné |
| IT | *Inter-commmit Time* | Temps moyen (en jours) entre les *commits* |
| LOC | *Lines of Code* | Nombre de lignes de code éditées par un développeur donné |
| C | *Complexity* | Complexité cyclomatique de McCabe [McC76] ajoutée ou supprimée par un développeur depuis le début du projet |
| SM | *Structural Modifications* | Modifications du code source effectuées par un développeur affectant la structure du code. Ces modifications incluent : les définitions de classes et de fonctions, les conditions, les boucles, etc. |
| NSM | *Non Structural Modifications* | Modifications du code source effectuées par un développeur n'affectant pas la structure du code. Ces modifications comprennent : les définitions de variables, les inclusions, les affectations, etc. |
| CM | *Comment* | Changement fait par un développeur sur les commentaires |
| NF | *Number of Files* | Nombre de fichiers édités par un développeurs |

TABLE I.3.8 – Variables utilisées par Di Bella *et al.* [DBSS13] pour le regroupement de développeurs.

- les développeurs clefs (*core developers*) qui développent la majeure partie du code et contribuent au projet sur un temps long,
- les développeurs actifs (*active developers*) qui développent une partie limitée du projet sur un temps long,
- les développeurs occasionnels (*occasional developers*) qui fournissent une contribution limitée sur une période de temps restreinte,
- les développeurs rares (*rare developers*) qui fournissent de petites contributions sur de très courtes périodes du projet.

Di Bella *et al.* utilisent l'apprentissage non-supervisé afin de former les quatre catégories de développeurs évoquées ci-dessus. Pour cela, ils ont extrait neuf variables (présentées en Table I.3.8) sur l'ensemble des développeurs présents dans neuf projets *open-source*.

Pour chacun des projets, Di Bella *et al.* utilisent la méthode des *k-means* pour grouper les développeurs en fonction des valeurs des neuf variables qui leur sont propres. Ainsi, Di Bella *et al.* montrent par une analyse empirique et statistique la possibilité de grouper les développeurs sur la base de ces neuf variables mais également la pertinence des quatre catégories de développeurs. De plus, ils ont également montré l'importance de certaines variables pour caractériser des catégories de développeurs comme le nombre de lignes de code, la complexité ou encore le temps moyen entre deux *commits*. Du fait de leur approche par apprentissage non-supervisé et de l'utilisation de métriques non-architecturales, il n'est pas possible, avec cette approche, d'identifier spécifiquement des développeurs ayant des compétences architecturales.

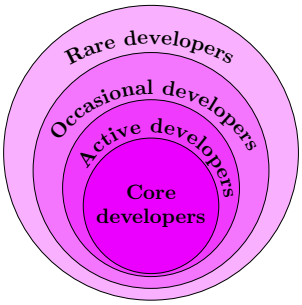



FIGURE I.3.2 – Catégories de développeurs créées par Di Bella *et al.* [DBSS13].

### 3.3.2 Approches par profilage

Les approches par profilages permettent la détection de développeurs expérimentés ou experts d'une ou plusieurs technologies mises en œuvre dans un projet logiciel.

#### 3.3.2.1 Expertise Browser

En 2002, Mockus *et al.* sont les premiers à proposer une solution permettant de mesurer l'expertise. Cette approche, nommée Expertise Browser [MH02], s'appuie sur le concept d'atomes d'expérience (*Experience Atoms*). Ces atomes d'expérience représentent des changements atomiques (ajout, suppression, modification) dans un fichier ou une documentation. C'est la somme de ces atomes qui permet de mesurer l'expérience d'un développeur, d'une équipe ou d'une organisation sur tout ou partie du projet. La mesure de l'expertise au travers de ces unités atomiques est faite depuis différentes sources de données : les fichiers de code source, le système de gestion de versions mais également la documentation. Par association des atomes d'expérience et des différents *artefacts* du projet, Expertise Browser permet de définir des niveaux d'expertise par élément ou partie du projet pour chacun des développeurs. Mockus *et al.* ont implémenté un prototype permettant d'utiliser l'approche Expertise Browser. Ce prototype permet de naviguer de manière arborescente dans les différents éléments du projet afin d'en distinguer des experts mais aussi de filtrer par développeur afin de visualiser les parties sur lesquelles un développeur particulier a travaillé. Notons que Mockus *et al.* ont également validé leur approche dans un contexte industriel avec 120 développeurs.

#### 3.3.2.2 Sindhgatta

Sindhgatta [Sin08] a proposé une approche utilisant le code source mais également les informations et *logs* fournis par les systèmes de gestion de versions. Sindhgatta extrait les différents domaines de compétences à maîtriser dans un projet à l'aide d'une approche non-supervisée. Comme illustré par la Figure I.3.3, Sindhgatta extrait les différents termes présents dans le code source puis les vectorise à l'aide d'une mesure de similarité. Les différents termes extraits sont regroupés à l'aide d'une méthode des *k-means*. Ces regroupements forment ainsi des groupes de compétences (sécurité, parallélisme, etc.) appartenant au projet. En parallèle, les *logs* du projet sont extraits

du système de gestion de versions afin d'identifier les fichiers sur lesquels ont travaillé chacun des développeurs. Ils sont ensuite rapprochés des domaines de compétences créés précédemment afin de mesurer la compétence du développeur pour chacun des domaines. Un des avantages liés à l'approche de Sindhgatta est la possibilité d'identifier de potentiels développeurs ayant des compétences architecturales. De plus, l'approche a été évaluée sur deux projets *open-source*.

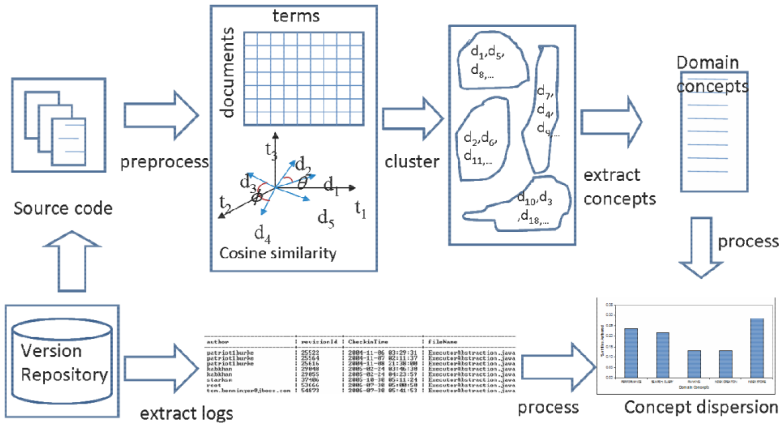



FIGURE I.3.3 – Processus d'extraction et d'identification des domaines de compétences des développeurs de Sindhgatta [Sin08].

#### 3.3.2.3 Schuler et Zimmermann

Schuler et Zimmermann [SZ08] ont proposé une approche relativement simple pour détecter les experts dans un projet logiciel. À l'aide des données issues du système de gestion de versions, Schuler et Zimmermann extraient les différentes modifications (ajout, suppression et modification) sur les méthodes du projet. Au niveau du code source, les auteurs utilisent un arbre syntaxique afin de rapprocher les modifications extraites de l'implémentation ou de l'usage des méthodes dans les sources. Schuler et Zimmermann distinguent ainsi deux types d'expertise :

- une expertise d'usage liée au fait d'utiliser un grand nombre de méthodes issues d'une même bibliothèque ou *package*,
- une expertise d'implémentation liée au fait de faire un grand nombre d'ajouts, de suppressions et de modifications de méthodes et de corps de méthodes.

De cette manière, Schuler et Zimmermann sont en mesure de distinguer des experts par technologie et de fait, de profiler des développeurs ayant des compétences architecturales. L'approche de Schuler et Zimmermann a été expérimentée sur un seul et unique projet *open-source*.

#### 3.3.2.4 LIBTIC

L'approche LIBTIC a été développée par Teyton *et al.* [TFMB13] pour détecter des experts technologiques et plus particulièrement des experts dans l'utilisation de librairies. Le principe de LIBTIC

consiste à associer à chaque développeur l'usage des éléments (méthodes ou attributs) appartenant aux différentes librairies composant le projet. Ainsi, connaissant l'ensemble des bibliothèques et leurs éléments, il est possible de calculer un taux d'expertise pour chaque développeur mais aussi pour chaque librairie. LIBTIC permet, en outre, la recherche d'experts d'une ou plusieurs librairies et ainsi le profilage d'experts ayant des connaissances sur des *frameworks* architecturaux. LIBTIC a été expérimentée sur un unique projet *open-source*.

#### 3.3.2.5 XTic

XTic est une approche développée par Teyton *et al.* [TPF+14] permettant de profiler des développeurs en fonction de compétences spécifiques décrites grâce à un *Domain Specific Language* (DSL). XTic se base sur le concept de changements atomiques de fichiers. Ce concept représente l'ensemble des modifications (ajouts, suppressions, modifications, renommages) faites par un développeur sur un fichier du projet. Chaque développeur du projet possède une collection de changements atomiques. C'est au travers de la collection des changements atomiques des développeurs qu'est réalisée la recherche de leurs compétences. La recherche des compétences des développeurs est effectuée grâce à des patrons syntaxiques. Ces patrons sont composés de quatre filtres permettant de décrire les compétences à rechercher :

- le filtre de chemin (*path filter*) permet filtrer la liste des fichiers utilisés pour extraire les compétences d'un développeur en spécifiant les chemins d'accès valides. Ce type de filtre permet, par exemple, de limiter la recherche à des types de fichiers spécifiques,
- le filtre du type de modification (*kind filter*) permet de filtrer le type de modifications effectuées sur les fichiers par un développeur,
- le filtre de contenu (*content filter*) permet de décrire le contenu que le développeur doit avoir ajouté dans les fichiers du projet. Le filtrage est fait par le biais d'expressions régulières,
- le filtre de modification de l'arbre syntaxique (*tree modification filter*) permet de sélectionner, au sein d'un arbre syntaxique abstrait les types de nœuds modifiés par un développeur.

```
<skill id="Creation of JUnit tests">
  <kind value="added"/>
  <files>
    <file value="\.java$"/></files>
  <files/>
  <contents>
    <content value="import org.junit" />
  </contents>
  <tree parser="java">
    <queries>
      <query value="//MethodDeclaration[MarkerAnnotation[@added]
                    /SimpleName[@label='Test'][@added]]"
    </queries>
  </tree>
</skill>
```

Listing I.3.1 – Exemple d'utilisation du *DSL* de XTic [TPF+14].

Le Listing I.3.1 présente un exemple de patron syntaxique permettant de filtrer les développeurs ayant travaillé sur des fichiers Java contenant la bibliothèque JUnit et ayant ajouté des méthodes

contenant le mot "Test" dans leurs déclarations. La flexibilité et la polyvalence offerte par XTic permet de profiler de manière précise des développeurs ayant des compétences architecturales sans pour autant se limiter à des technologies ou des bibliothèques. Notons également que l'approche dénombre le nombre de correspondances entre les compétences spécifiées et les changements atomiques ce qui permet, par la suite, le regroupement des développeurs en catégories représentant leur plus ou moins grande expérience. Teyton *et al.* ont validé leur approche sur un corpus de seize projets Java issus de GitHub mais également sur un projet provenant d'un industriel.

#### 3.3.2.6 Hauff et Gousios

Hauff et Gousios [HG15] ont développé une approche pour faire correspondre des profils de développeurs à des offres d'emploi en utilisant des fichiers ReadMe et des méta-données GitHub. Cette approche est illustrée par la Figure I.3.4. À la différence des approches décrites précédemment, celle de Hauff et Gousios [HG15] vient en rupture par la non utilisation du code comme source de données. Pour cela, les auteurs extraient des concepts ontologiques correspondant aux compétences recherchées depuis des offres d'emploi. Le même type d'extraction est réalisé sur les fichiers ReadMe dans les dépôts de code GitHub d'un développeur donné. L'ensemble des concepts extraits des offres d'emplois et des contributions des développeurs est pondéré. Les compétences des développeurs sont ensuite associées aux offres d'emploi en reliant les concepts ayant la même pondération dans l'offre et dans les fichiers ReadMe analysés.

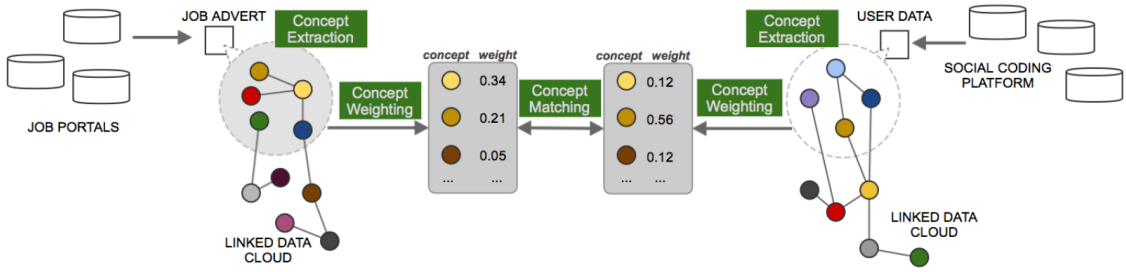



Figure I.3.4 – Processus de Hauff et Gousios [HG15] d'extraction des données et de mise en relation entre un développeur et une offre d'emploi.

#### 3.3.2.7 CVExplorer

CVExplorer [GF16] est une approche qui s'inscrit dans la même veine que les travaux de Hauff et Gousios [HG15] à savoir le profilage des compétences de développeurs en vue d'évaluer leur adéquation à un emploi. L'approche de CVExplorer est présentée Figure I.3.5. Comme Hauff et Gousios [HG15], CVExplorer n'utilise pas le code source mais les méta-données GitHub et les fichiers ReadMe. L'approche génère un treillis de concepts [BM70, GW99]. La visualisation des nœuds du treillis est réalisée à l'aide d'un nuage de mots. L'utilisateur peut sélectionner des mots clefs pour naviguer dans le treillis et affiner un profil de développeur en fonction de ses besoins.

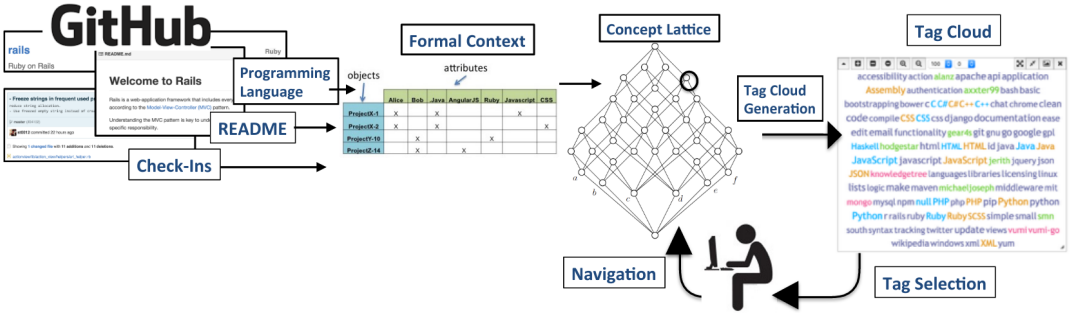



FIGURE I.3.5 – Processus d'extraction des compétences des développeurs de CVExplorer [GF16].

#### 3.3.2.8 Santos *et al.*

Santos *et al.* [SdASOF18] ont, à l'image de LIBTIC [TFMB13], cherché à identifier des experts dans l'utilisation de bibliothèques. Pour cela, Santos *et al.* définissent cinq dimensions de compétences liées aux développeurs :

- l'amplitude de connaissance mesure l'étendue des connaissances d'un développeurs sur différentes bibliothèques. Elle est mesurée par le nombre d'imports uniques réalisés par un développeur pour chaque bibliothèque,
- l'intensité de la connaissance mesure l'intensité d'utilisation d'une bibliothèque. Pour mesurer cette dimension, l'ensemble des imports faits par un développeur pour une bibliothèque sont dénombrés,
- la maîtrise du code mesure la productivité d'un développeur pour une bibliothèque donnée, en utilisant le nombre de lignes ajoutées liées à l'import d'une bibliothèque.
- la collaboration mesure la capacité d'un développeur à collaborer avec les autres développeurs. Elle est mesurée en comptant le nombre de fichiers de code source que le développeur a modifié avec d'autres pairs.
- l'expérience : mesure le nombre de projets additionnels dans lesquels le développeur utilise une bibliothèque donnée.

Chacune des dimensions est mesurée à l'aide de métriques liées au code et à des méta-données (nombre de projets, nombre de lignes de code, nombre d'imports). Pour chacune de ces métriques, des seuils sont définis de sorte à créer cinq catégories de développeurs. Ainsi, l'approche de Santos *et al.* permet à la fois de grouper les développeurs en cinq catégories mais également de profiler des développeurs en fonction de leur niveau de compétence dans chaque dimension. Elle permet également de profiler des développeurs ayant des compétences architecturales sous réserve d'utilisation d'un *framework* de gestion d'architectures le projet. Une large évaluation de l'approche de Santos *et al.* a été menée sur près de 35000 projets provenant de GitHub.

### 3.3.3 Discussion

Les approches par profilage de développeurs ont été les premières développées [MH02, Sin08, SZ08]. Elles utilisent pour la plupart le code source ainsi que des données issues des logiciels de gestion de versions [MH02, Sin08, SZ08, TFMB13, TPF+14, SdASOF18] pour détecter les compétences des développeurs. Ces approches ont, de plus, l'avantage de ne pas utiliser d'artefacts externes au code comme de la documentation. Deux autres approches de profilage [HG15, GF16] n'utilisent pas directement le code source mais des méta-données GitHub et des fichiers ReadMe. Ces approches sont moins fiables que celles utilisant le code source car elles se basent sur les déclarations des développeurs dans les ReadMe. Cette fiabilité est d'autant plus mise à mal que ces déclarations peuvent être faites par d'autres développeurs que le propriétaire du dépôt logiciel. Parmi l'ensemble des approches, seules trois [TPF+14, GF16, SdASOF18] sont capables de profiler des développeurs avec des compétences architecturales. XTic [TPF+14] utilise un DSL pour spécifier les compétences recherchées. L'approche CVExplorer de Hauff et Gousios [HG15] permet de profiler un développeur par parcours d'un treillis et permet donc de profiler plus spécifiquement des compétences architecturales tandis que l'approche de Santos *et al.* [SdASOF18] permet de mesurer la compétence d'un développeur sur une technologie spécifique, donc potentiellement un *framework* architectural.
Globalement, les approches par regroupement [KHM08, DBSS13] ne proposent pas de distinguer spécifiquement les développeurs avec des compétences architecturales. Ces approches se basent sur des éléments structurels [KHM08] ou des métriques [DBSS13] générales, mesurant un niveau d'implication et de responsabilité, plutôt qu'une compétence particulière. Dans le panel d'approches que nous comparons, seule XTic [TPF+14] permet à la fois de profiler (compétences architecturales) et regrouper des développeurs en fonction de leurs compétences. Une des spécificités qui fait la polyvalence de XTic est la description via un DSL des compétences recherchées.

## 3.4 Conclusion

Dans ce chapitre, nous avons proposé l'état de l'art concernant trois types de travaux.
Concernant la classification de tickets nous avons vu qu'il existait différents approches et jeux de données. Parmi les quatre jeux recensés [AAP+08, HJZ13, OAH19, KSCP19], celui de Herzig *et al.* [HJZ13] est disponible en ligne et manuellement étiqueté, ce qui renforce sa fiabilité. Nous avons vu également qu'il existe dix approches de classification sur ces jeux de données dont six [PHM13, CS15, PSHS17, THPM17, QS18, PSHS17] utilisant celui de Herzig *et al.*. Ces approches utilisent des techniques de classification variées parmi lesquelles on retrouve *Random Forest*, *Decision Tree*, *Support Vector Machine*, *k-Neareast Neighbors*, *Linear Discriminant Analysis*, *Naive Bayes*. Du fait de la mise en ligne du jeu de données de Herzig *et al.* et de sa réutilisation par six travaux, nous travaillerons avec ce jeu de données dans la suite de cette thèse. Les approches de classification comparent pour leur majorité, différents types de classifieurs. Pour répondre à la Question de recherche 2 nous allons nous appuyer sur les travaux qui concernent le jeu de données de Herzig *et*

*al.* et comparer un panel de classifieurs.
Concernant les approches permettant la mesure de la contribution des développeurs. Nous n'avons trouvé que trois travaux autour de cette thématique [MH02, BNM+11, FFB14]. Les travaux de Bird *et al.* [BNM+11] ont permis la description de la mesure de la contribution ainsi que la définition d'un seuil de contribution permettant la catégorisation des développeurs. Foucault *et al.* [FFB14] ont formalisé cette contribution et l'ont expérimenté sur un panel de projets *open-source*. Nous remarquons qu'il n'existe pas de métrique permettant l'évaluation de la contribution des développeurs à l'architecture logicielle. Ainsi pour répondre à la Question de recherche 1, nous allons reprendre les travaux de Bird *et al.* [BNM+11] et notamment le seuil permettant de catégoriser les développeurs en fonction de leur taux de contribution. Nous allons également nous appuyé sur la formalisation de Foucault *et al.* [FFB14] pour créer notre métrique de contribution et répondre à la Question de recherche 1.
Enfin, nous avons recensé les approches de détection de l'expertise logicielle, nous avons recensé neuf travaux. Nous pouvons scinder ces approches en deux groupes : celles permettant le regroupement de développeurs en fonction de caractéristiques et celles dédiées au profilage de développeurs. Nous avons vu que les approches par regroupement [TPF+14, DBSS13, KHM08] utilisent des méthodes d'apprentissage non-supervisées et des données extraites du code source et des systèmes de gestion de versions. Les approches de profilage [SdASOF18, GF16, HG15, TPF+14, TFMB13, SZ08, Sin08, MH02] utilisent des stratégies et des sources données différentes (code source, méta-données, documentation, etc.). Certaines sont focalisées sur la correspondance entre développeurs et offres d'emploi [HG15, GF16] tandis que d'autres [SdASOF18, TPF+14, TFMB13, SZ08, Sin08, MH02] permettent le profilage d'une ou plusieurs compétences spécifiques. Nous remarquons que ces approches n'utilisent pas d'apprentissage supervisé. L'ensemble de ces approches montre que le profilage et/ou le regroupement peut avoir plusieurs objectifs : découverte de compétences particulières, correspondance à des offres d'emploi ou encore regroupement par catégories sur la base de métriques. À travers ces approches, et notamment sur celles utilisant des métriques, nous voyons la nécessité de prendre en compte la pluralité des compétences des développeurs (qualitativement ou quantitativement) pour répondre à la Question de recherche 3.

# Deuxième partie

# Contributions

# Chapitre II.4

# Classification automatique de tickets de bogues

**Sommaire**

## 4.1 Introduction

Le terme de bogue a été employé pour la première fois par Thomas Edison en 1873 alors qu'il travaillait à la conception d'un système télégraphique permettant de transmettre et de recevoir simultanément jusqu'à quatre télégrammes distincts sur un seul fil. Ce terme a par la suite été utilisé par Grace Hopper lors d'une panne due à un insecte (en anglais *bug*) coincé entre deux contacts d'un relais de l'ordinateur électromécanique Harvard Mark II. Cet incident a été consigné dans le rapport de maintenance de la machine et peut donc être considéré comme l'un des premiers rapports de bogue survenant sur un ordinateur. Aujourd'hui les rapports d'incidents logiciels aussi appelés tickets de bogues (ou *bug tickets*) sont gérés par des systèmes de suivi de tickets ou *issue tracking systems* (IST). Les tickets logiciels sont des éléments textuels contenant un titre, une description et auxquels est attaché un type. Les types de tickets peuvent-être variés. Parmi les types les plus courants on retrouve : les questions sur le code, les demandes d'amélioration, les descriptions de bogues... Les IST ont la particularité de pouvoir gérer le cycle de vie du ticket mais également d'assigner un type spécifique au ticket. Ainsi il est possible d'identifier le type d'un ticket en vue de son orientation vers un développeur spécifique ou pour connaître le nombre de tickets d'un type spécifique à un instant $t$. L'émergence des IST a soulevé un problème de volumétrie des tickets sur de grands projets. En effet, les développeurs de Mozilla rapportent dans l'étude de Anvik *et al.* [AHM05] que "Chaque jour, environ 300 bogues sont rapportés et ont besoin d'être assignés à un ou plusieurs développeurs. Cela représente un volume que les développeurs de Mozilla seuls ne peuvent gérer". Un autre aspect important vient du coût engendré par les tickets. Une étude de l'Université de Cambridge [BJC+13] estime à 312 milliards de dollars le coût du débogage (salaires de développeurs) par an. Notons également qu'il existe une corrélation entre bogues et méthodes de management, étudiée par Bachmann et Bernstein [BB10]. Ainsi savoir catégoriser de manière correcte les tickets permet une réduction des coûts humains et financiers. Cela peut également alimenter des indicateurs de pilotage dans le cadre de processus qualité, notamment dans le cas des tickets de bogues.

L'objectif de ce chapitre est de présenter une approche permettant de classer de manière automatique les tickets dans deux catégories : bogues ou non. Ce travail est une première étape (preuve de concept) avant d'appliquer cette approche à d'autres types de tickets et plus particulièrement ceux liés à l'architecture logicielle. La Section 4.3 décrit l'approche globale que nous proposons pour la classification de tickets. La Section 4.4 détaille cette approche. Puis la Section 4.5 présente et commente les résultats obtenus. La Section 4.6 confronte notre classifieur à des tickets issus de trois projets *open-source*. La Section 4.7 présente un prototype d'assistance à la rédaction de tickets qui utilisent le classifieur précédemment créé. Enfin la Section 4.8 présente les limites et perspectives de ces travaux.

## 4.2 Questions de recherche

Nous avons vu en introduction de ce chapitre que les tickets et plus particulièrement ceux décrivant des bogues étaient des éléments indissociables des projets logiciels. En Section 3.1 nous avons également vu les différentes approches de classification automatique de tickets et les résultats obtenus. Les systèmes d'assignation automatique de types de tickets ont été créés à cause de l'augmentation de la volumétrie des tickets sur les projets de tailles importantes [ZTGG16]. Les *issue tracking systems* (ITS), au travers de leurs API, permettent de récupérer des ensembles importants de tickets pour alimenter la recherche empirique en génie logiciel [HJZ13]. Cependant, des erreurs de classification ou un non étiquetage peuvent engendrer des biais dans les études. En effet, Herzig *et al.* [HJZ13] ont montré qu'environ 30% des tickets étaient victimes d'erreurs de classification. Sur la base de ces éléments, nous formulons deux questions de recherche :

**Question de recherche 4.1**

Est-il possible de créer un classifieur suffisamment performant (supérieur à l'état de l'art) permettant de classer des tickets dans deux catégories : bogue ou non bogue ?

La Question de recherche 4.1 interroge la possibilité de disposer d'un classifieur assez performant pour assurer l'automatisation de tâches reposant sur le tri des tickets dans les outils assistant les processus de développement ou dans un cadre de recherche empirique.

**Question de recherche 4.2**

Qu'elle est la généricité d'un classifieur entraîné sur un ensemble de tickets portant sur un langage spécifique ?

La Question de recherche 4.2 questionne la généricité d'un classifieur entraîné sur un jeu de données contenant des tickets portant sur un seul langage et sur un nombre limité de projets (nettoyage manuel). Ce type de jeu de données permet d'évaluer la performance pure du classifieur, sur un contexte restreint. La question de son extension à de multiples projets et de multiples langages est un aspect à étudier afin de déterminer la stratégie à adopter : entraînement de classifieurs génériques ou entraînement de classifieurs spécifiques (avec ou sans *transfer learning*).

## 4.3 Approche globale

Dans cette section, nous fournissons une vision globale de l'approche mise en place dans le but de classifier automatiquement des tickets de bogues. Comme illustré Figure II.4.1, notre approche est composée de cinq grandes étapes que nous allons décrire dans cette section.

### Extraction du corpus de tickets

La première étape consiste à extraire un corpus de tickets. Pour cela nous allons utiliser le corpus nettoyé et manuellement annoté par Herzig *et al.* [HJZ13]. Herzig *et al.* ont mis en ligne [1] l'ensemble des identifiants des tickets utilisés dans leur étude ainsi que les étiquettes à associer aux tickets. Nous utilisons les API HTTP fournies par l'ITS Jira pour récupérer le corpus de tickets proposé par Herzig *et al.* Les tickets sont ensuite associés à leurs étiquettes puis stockés dans un fichier JSON unique.

### Traitement textuel du corpus

La deuxième étape consiste à traiter les informations textuelles du corpus. Les données textuelles sont filtrées et nettoyées afin de ne garder que les informations pertinentes. Pour cela nous faisons appel à des techniques de traitement du langage naturel (TALN). L'ensemble du texte du ticket subit une tokenisation. Cette étape permet le découpage du texte des tickets en petites unités appelées *tokens*. Ensuite, par fenêtre glissante, des tuples de *tokens* successifs appelés $n$-grammes sont construits. La fréquence de chacun de ces $n$-grammes est calculée puis les $n$-grammes sur-représentés et sous-représentés sont éliminés. Les éléments sur-représentés sont appelés *stop-words* et correspondent souvent à des articles comme : "a", "an", "the", "or", etc. Les éléments sous-représentés sont ceux n'apparaissant qu'une à deux fois dans un document. Enfin, les informations textuelles sont converties en informations mathématiques. Pour cela, chaque ticket est transformé en un vecteur contenant la fréquence d'apparition, dans son contenu, des différents $n$-grammes existant dans le corpus des tickets.

1. https://www.st.cs.uni-saarland.de/softevo/bugclassify/

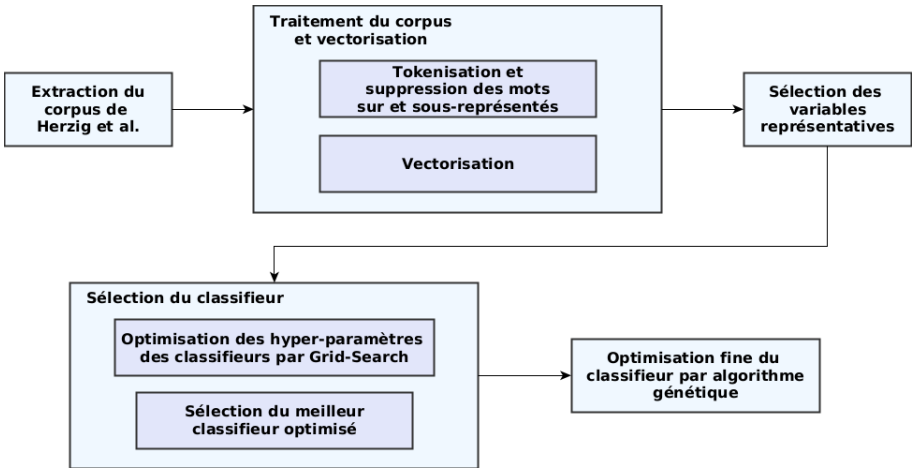



FIGURE II.4.1 – Vue globale du processus de classification de tickets de bogues.

### Sélection des éléments textuels représentatifs des tickets

Les vecteurs représentant les données textuelles ont une grande dimension bien que toutes les fréquences des $n$-grammes n'aient pas la même importance pour la classification des tickets. Une sélection des $n$-grammes les plus représentatifs du corpus est effectuée afin de réduire la dimension des vecteurs. L'objectif est d'améliorer la performance de l'apprentissage par la machine à la fois sur le plan du temps de calcul mais aussi sur celui de la précision (au sens général) de la classification.

### Sélection d'un classifieur

Une quatrième étape consiste à sélectionner un classifieur. Disposant d'un panel de classifieurs, choisir le bon est une étape importante afin d'avoir de meilleures performances. Notre jeu de données (corpus de tickets) étant annoté, nous explorons des techniques d'apprentissage supervisé. Pour cela nous allons comparer six classifieurs :

- ***Stochastic Gradient Descent (SGD)*** utilise la méthode de descente du gradient pour minimiser une fonction objectif écrite comme une somme de fonctions :

$$Q(w) = \frac{1}{n}\sum_{i=1}^{n} Q_i(w)$$

  où $w$ est le paramètre à estimer dans le but de minimiser la fonction $Q(w)$. $Q_i$ correspond à la $i$-ième observation dans le jeu de données d'entraînement.

- ***Support Vector Machines (SVM)*** sont des types de classifieurs non-probabilistes basés sur l'algèbre linéaire. L'entraînement des SVMs crée des hyperplans capables de séparer des données multidimensionnelles dans différentes classes. Les SVMs optimisent la position des hyperplans en maximisant la distance avec les données les plus proches. De fait, ce type de classifieur a généralement une bonne justesse.

- ***Random Forest (RF)*** est une méthode d'apprentissage parallèle basée sur une forêt d'arbres décisionnels créée aléatoirement. Chaque arbre de la forêt est entraîné sur un sous-ensemble de forêts aléatoires en utilisant le principe du *bagging*. Le *bagging* permet la réduction de la complexité du problème en le distribuant sur des sous-ensemble de forêts.

- ***Ridge Regression (RR)*** est basée sur une méthode des moindres carrés avec une régularisation *L2* (aussi connue sous le nom de régularisation de Tikhonov): une pénalité égale à la somme des carrés des poids, multipliée par un facteur de pénalité $\alpha$ est ajoutée à la fonction objectif à minimiser $f_{loss}$. Ainsi la fonction objectif de RR est définie comme suit :

$$Obj = f_{loss}(w) + \alpha \cdot \sum w^2$$

  $f_{loss}$ dépend de la tâche sous-jacente, à savoir la perte d'entropie croisée pour la classification. $\alpha$ est généralement ajusté durant la validation du modèle et est appelé paramètre de régularisation.

- ***k-Nearest Neighbors (kNN)*** est une méthode non-paramétrique dans laquelle le modèle stocke les données d'entraînement pour effectuer la classification des données de test. Pour classifier une instance, kNN recherche les $k$ plus proches voisins de l'instance donnée en entrée en fonction de leurs poids ou de leurs distances selon la méthode utilisée. De cette manière kNN attribue une classe à l'instance à classifier.
- ***Multi-Layer Perceptron (MLP)*** est un type de réseau de neurones formels organisé en plusieurs couches. Le flux d'informations parcourt l'ensemble du réseau en allant de la couche de neurones d'entrée à la couche de sortie par le biais de connexions pondérées. L'entraînement supervisé ajuste de manière incrémentale les poids sur les connexions des neurones par rétro-propagation de l'erreur. Cet ajustement est fait jusqu'à ce que la sortie du MLP soit correcte. L'utilisation de plusieurs couches rend le MLP capable de classifier des données non linéairement séparables.

Les classifieurs disposent de paramètres appelés hyper-paramètres qui influencent l'apprentissage et donc, *in fine*, leurs performances. Trouver la bonne combinaison d'hyper-paramètres permet donc de meilleures performances de classification. Nous utilisons une méthode combinatoire afin d'optimiser chaque classifieur avant de les comparer entre-eux et de sélectionner le meilleur.

### Optimisation par algorithme génétique du classifieur

Le classifieur MLP ayant donné les meilleurs performances, nous le sélectionnons et optimisons plus finement ses paramètres à l'aide d'un algorithme génétique. L'optimisation concerne le nombre de *features* (variables) données en entrée au MLP. Ce nombre de *features* correspond à la taille de la couche de neurones d'entrée. Nous optimisons également le nombre et les tailles des couches cachées. Sur ce type de paramètres, le gain espéré avec des méthodes discrètes (dichotomiques ou combinatoires) est moindre qu'avec une méthode capable d'explorer un espace de solutions plus grand, comme un algorithme génétique peut le faire.

## 4.4 Approche détaillée

Dans la Section 4.3, nous avons montré les grandes étapes de l'approche que nous avons mis en place pour classifier des tickets de bogues. Dans cette section nous détaillons l'ensemble des étapes nous permettant d'aboutir à une classification performante des tickets de bogues en utilisant un classifieur du type *Multi-Layer Perceptron*.

### 4.4.1 Extraction des tickets de bogues

Les données utilisées pour la classification proviennent du corpus de tickets de Herzig *et al.* [HJZ13] (voir Section 3.1). Nous avons extrait 5591 tickets de trois grands projets Java *open-source*: *Lucene*, *JackRabbit* et *HttpClient*. Après extraction des tickets, ceux-ci sont associés à leur classification ("bug" / "non bug"). Cette association est réalisée à l'aide de l'identifiant du ticket et de l'étiquette

```
{
  "key": "HTTPCLIENT-126",
  "summary": "Default charset",
  "description": "As defined in RFC2616 the default character set is
      ISO-8859-1 an not US-ASCII \nas defined in HttpMethodBase. See
      \"3.7.1 Canonicalization and Text Defaults\" at\nRFC 2616",
  "classified": "IMPROVEMENT",
  "type": "BUG",
  "label": "NBUG"
}
```

Listing II.4.1 – Exemple de ticket de bogue utilisé pour tester et entraîner le classifieur.

donnée dans le corpus de Herzig *et al.*. Enfin, l'ensemble des tickets et de leurs étiquettes est stocké dans un unique fichier JSON. Un exemple de ticket annoté est donné par le Listing II.4.1. La structure JSON est composée des six champs suivants :

- `key` représente l'identifiant unique du ticket dans le système de suivi de tickets. C'est cet identifiant qui permet la correspondance entre un ticket et l'étiquette définie par Herzig *et al.*.
- `summary` est la description courte (titre) du ticket.
- `description` est la description longue du ticket. Pour un ticket de bogue par exemple, cette description contient généralement les causes et les problèmes engendrés.
- `classification` est la catégorie du ticket (étiquette) donnée par Herzig *et al.*.
- `type` est la catégorie du ticket mentionnée par l'auteur du ticket dans le système de suivi.
- `label` est l'étiquette que nous avons définie pour le ticket : `BUG` pour un ticket classé par Herzig *et al.* comme un bogue (dans le champ `classification`). Les autres catégories sont classées `NBUG`.

### 4.4.2 Traitement textuel du corpus

Après l'extraction, un traitement du type *bag-of-word* est réalisé afin de convertir l'information textuelle des tickets en informations vectorielles que les classifieurs peuvent utiliser. Cette étape a un impact sur la performance du classifieur. Pour cela, nous utilisons l'API de Scikit-learn [BLB+13]. Cette API d'apprentissage automatique a fait ses preuves dans de nombreux projets de laboratoires ou industriels [DKD21] de par son caractère standardisé, configurable et vaste en terme de méthodes fournies. Comme montré par la Figure II.4.2, cette étape de traitement des informations textuelles est divisée en deux parties : le traitement du langage naturel puis sa vectorisation et la sélection des variables les plus représentatives (plus communément appelé *feature selection*).

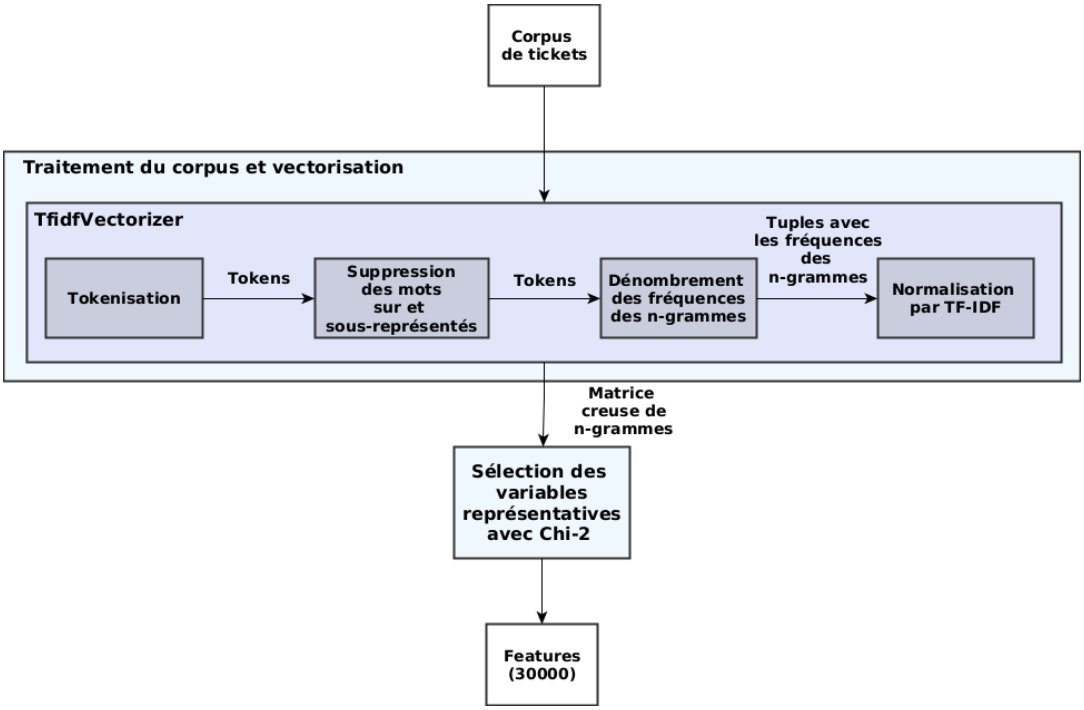



FIGURE II.4.2 – Vue globale du processus de classification de tickets de bogues.

#### 4.4.2.1 Traitement du langage naturel

Dans cette étape, le corpus est nettoyé puis transformé en vecteurs utilisables par un classifieur. Nous calculons des valeurs statistiques sur les éléments textuels en utilisant la méthode *Term Frequency Inverse Document Frequency (TF-IDF)* [Jon72] puis assemblons ces éléments en vecteurs. Pour réaliser ces étapes nous utilisons la méthode `TfidfVectorizer` implémentée dans Scikit-learn.

1. **Tokenisation.** Les données textuelles sont divisées en unités de texte appelées *tokens* où chaque *token tok* représente un mot. À partir de ces *tokens*, des $n$-grammes sont générés. Un $n$-gramme est une séquence de $n$ *token* contigus. Un $n$-gramme peut s'écrire formellement de la manière suivante : $n$-gramme $= \{tok_1, tok_2, ..., tok_n\}$. Dans nos travaux, nous utilisons trois types de $n$-grammes : des uni-grammes (Listing II.4.2), des bi-grammes (Listing II.4.3) et des tri-grammes (Listing II.4.4).

```
"see", "371", "canonicalization", "text", "defaults", "rfc", "2616"
```

Listing II.4.2 – Exemples d'uni-grammes créés avec la dernière phrase du Listing II.4.1.

```
"see 371", "371 canonicalization" , "canonicalization text", "text defaults",
"defaults rfc", "rfc 2616"
```

Listing II.4.3 – Exemples de bi-grammes créés avec la dernière phrase du Listing II.4.1.

```
"see 371 canonicalization", "371 canonicalization text", "canonicalization text defaults",
"text defaults rfc", "defaults rfc 2616"
```

Listing II.4.4 – Exemples de tri-grammes créés avec la dernière phrase du Listing II.4.1.

2. **Suppression des mots sur-représentés (*stop-words*) et sous-représentés.** Les *tokens* ayant une fréquence d'occurrence élevée dans le corpus (*stop-words*) sont souvent peu pertinents dans la classification textuelle [SFHA14]. C'est pourquoi, les *tokens* apparaissant dans plus de 50% des tickets du corpus sont filtrés. À l'inverse, les *tokens* avec de faibles taux d'occurrences sont aussi peu pertinents. Ainsi, les *tokens* apparaissant dans moins de 2% des tickets du corpus sont également filtrés. Un ticket contient deux informations principales : un titre et une description. Un exemple de titre et de description est donné par le Listing II.4.1. Le titre est une sorte de résumé du ticket tandis que la description est l'explication précise du ticket. Des études [PSHS17] utilisent seulement le titre du ticket et ignorent la description. Cependant, les résultats de classification sont meilleurs quand titre et description sont utilisés ensemble [LSK+19, PHM13, CS15, THPM17]. D'autres travaux augmentent le poids du titre par la multiplication de son contenu [PSHS17, KMC06]. Afin de capitaliser sur ces études nous avons menés des tests de multiplication du titre. Il apparaît que multiplier le contenu du titre par trois donne de meilleures performances sur la classification.

3. **Calcul de la fréquence des $n$-grammes.** Les occurrences de chaque uni-gramme, bi-gramme et tri-gramme sont comptées. La résultante est un tuple pour chaque $n$-gramme avec sa fréquence dans chaque ticket.

4. **Normalisation par TF-IDF.** La méthode *Term Frequency - Inverse Document Frequency (TF-IDF)* est utilisée pour pondérer et normaliser les fréquences des $n$-grammes dans le corpus. Cette méthode statistique évalue l'importance d'un terme $t$ (un $n$-gramme) contenu dans un document $d$ (un ticket), relativement au corpus $D$ de documents (l'entièreté du corpus de tickets). Plus précisément, $tf$ permet de mesurer l'importance d'un terme $t$ en fonction de sa fréquence dans un document $d$. $tf$ est ensuite pondéré par $idf$ qui représente la fréquence d'apparition du terme $t$ dans l'ensemble du corpus $D$. La pondération diminue avec le nombre d'occurrences du terme $t$ dans le corpus $D$ ce qui permet de mettre en avant les termes dont la présence est discriminante.

   La formule de TF-IDF est la suivante :

$$tfidf(t, d, D) = tf(t, d) \cdot idf(t, D) \tag{II.4.1}$$

   Dans l'Équation (II.4.1), $tf$ correspond au nombre d'occurrences d'un terme $t$ dans un document $d$ (noté $n_{t,d}$) divisé par le total du nombre de termes dans le document $d$ (noté

$n_d$) :

$$tf(t,d) = 1 + \log\left(\frac{n_{t,d}}{\sum n_d}\right) \tag{II.4.2}$$

$tf$ est calculé en utilisant $1 + log$ pour limiter le poids des termes très fréquents.

$idf$ représente l'importance d'un terme $t$ dans le corpus $D$. Le nombre de documents où le terme $t$ apparaît dans le corpus $D$ est noté $n_{t,D}$ :

$$idf(t,D) = \log\left(\frac{1+|D|}{1+n_{t,D}}\right) + 1 \tag{II.4.3}$$

Dans l'Équation (II.4.3), $idf$ est défini dans sa version dite *smooth* en ajoutant 1 après la fonction $log$. La constante 1 est ajoutée au numérateur et au dénominateur pour prévenir les divisions par zéro.

Cette méthode est implémentée dans Scikit-learn sous le nom `TfidfVectorizer`. Elle retourne une matrice creuse avec les pondérations calculées par TF-IDF pour l'ensemble des $n$-grammes de chaque ticket du corpus.

#### 4.4.2.2 Sélection des variables

Une sélection des variables (*features*) les plus représentatives est réalisée avant l'entraînement du classifieur afin d'éviter les variables non-pertinentes ou amenant du bruit. En effet, initialement le nombre de variables est très important : 24496 en considérant seulement les uni-grammes, 63925 en tenant compte des uni-grammes et bi-grammes. Ce nombre augmente jusqu'à 99351 comptant les uni-grammes, les bi-grammes et les tri-grammes. Le nombre de variables sélectionnées influence le temps d'apprentissage, l'ajustement du modèle et la qualité du classifieur. Trouver un nombre de *features* adéquat est donc important.

Un test statistique du Chi-deux est utilisé pour mesurer la relation d'association entre nos variables catégorielles (`"BUG"` ou `"NBUG"`) et nos *features*. Le test du Chi-deux sélectionne les *features* les plus pertinentes pour chaque variable catégorielle par rapport au nombre de *features* maximum défini par l'utilisateur. Nous avons empiriquement fixé le nombre de *features* maximum à 30000. La détermination de cette valeur s'est faite par échantillonnage du nombre de *features* dans la plage $[5000 - 60000]$ par incrément de 5000. Pour chaque échantillon nous avons calculé la mesure F1 (en utilisant une évaluation 10-fold) pour chaque classifieur testé dans nos travaux. En dessous de 30000 *features*, les mesures F1 de l'ensemble des classifieurs indiquent de faibles performances. Au-delà de 30000, les mesures F1 des classifieurs testés ont trop de variabilité pour être facilement comparées. Il apparaît que 30000 est le meilleur compromis pour mesurer la performance globale des classifieurs. Le test du Chi-deux va donc sélectionner 30000 variables représentatives de la catégorie des tickets décrivant des bogues. Le nombre de *features* en entrée des classifieurs étant fixé, il est maintenant possible de les comparer et de sélectionner le plus performant.

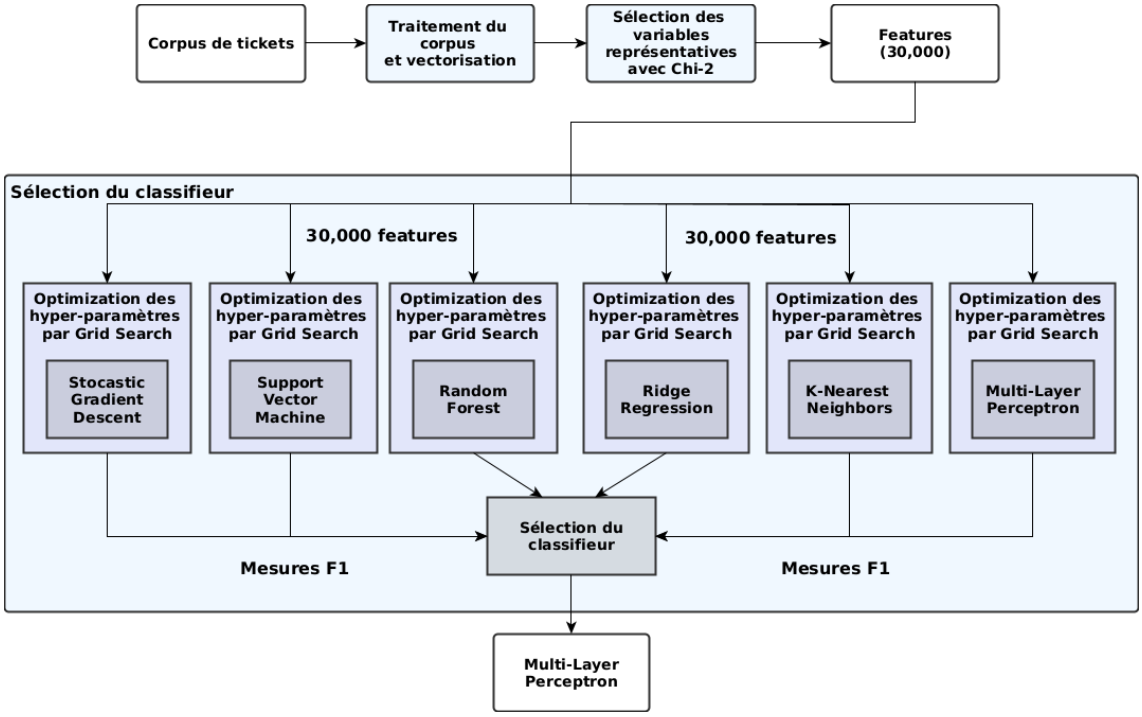



FIGURE II.4.3 – Processus de sélection du classifieur pour la classification de tickets.

### 4.4.3 Sélection des classifieurs

Dans les sections précédentes, nous avons mis en place le processus de traitement des informations textuelles et la conversion de celles-ci en vecteurs. Puis, nous avons sélectionné parmi ces vecteurs les plus représentatifs du corpus en fonction de nos besoins de classification. Nous allons donc maintenant sélectionner le classifieur le plus performant parmi un panel de six (décrits en Section 4.3). La Figure II.4.3 montre les six classifieurs que nous avons utilisés dans le processus de sélection. Comme nous l'avons mentionné en Section 4.3, ces classifieurs utilisent des hyper-paramètres influençant leurs performances. Ainsi, à type de classifieur égal, des hyper-paramètres corrects pourront avoir une impact important sur les performances d'où la nécessité de choisir une bonne combinaison d'hyper-paramètres. Nous utilisons un algorithme combinatoire appelé *Grid-Search* permettant de rechercher une combinaison optimisée d'hyper-paramètres. *Grid-Search* parcourt l'ensemble des combinaisons de valeurs générées par le produit Cartésien de $n$ ensembles de valeurs associées à chaque hyper-paramètre. Pour cela, un ensemble de valeurs discrètes à tester est fournie à *Grid-Search* pour chaque hyper-paramètres. Tous les $n$-tuples d'hyper-paramètres sont générés et utilisés pour initialiser les classifieurs. Puis les classifieurs sont entraînés avec 30000 *features* choisis grâce au chi-deux et évalués par 10-*fold*. Le *Multi-Layer Perceptron* donnant les meilleurs résultats (en termes de mesure F1), il est finalement sélectionné comme classifieur.

### 4.4.4 Optimisation par algorithme génétique

Comme nous l'avons dit précédemment, l'optimisation des hyper-paramètres est cruciale pour la qualité de la prédiction. Ceux-ci ne pouvant être optimisés finement à l'aide de *Grid-Search* (qui utilise des combinaisons de valeurs discrètes prédéfinies) nous avons mis en œuvre un algorithme génétique (AG) [Hol92] pour réaliser des recherches de solutions de bonne qualité. Nous utilisons l'algorithme génétique pour optimiser trois paramètres de notre MLP, à savoir :

- le nombre de *features* données en entrée du classifieur (équivalent à la taille du nombre de neurones sur la couche d'entrée),
- le nombre de couches cachées de neurones,
- la taille de chacune des couches cachées.

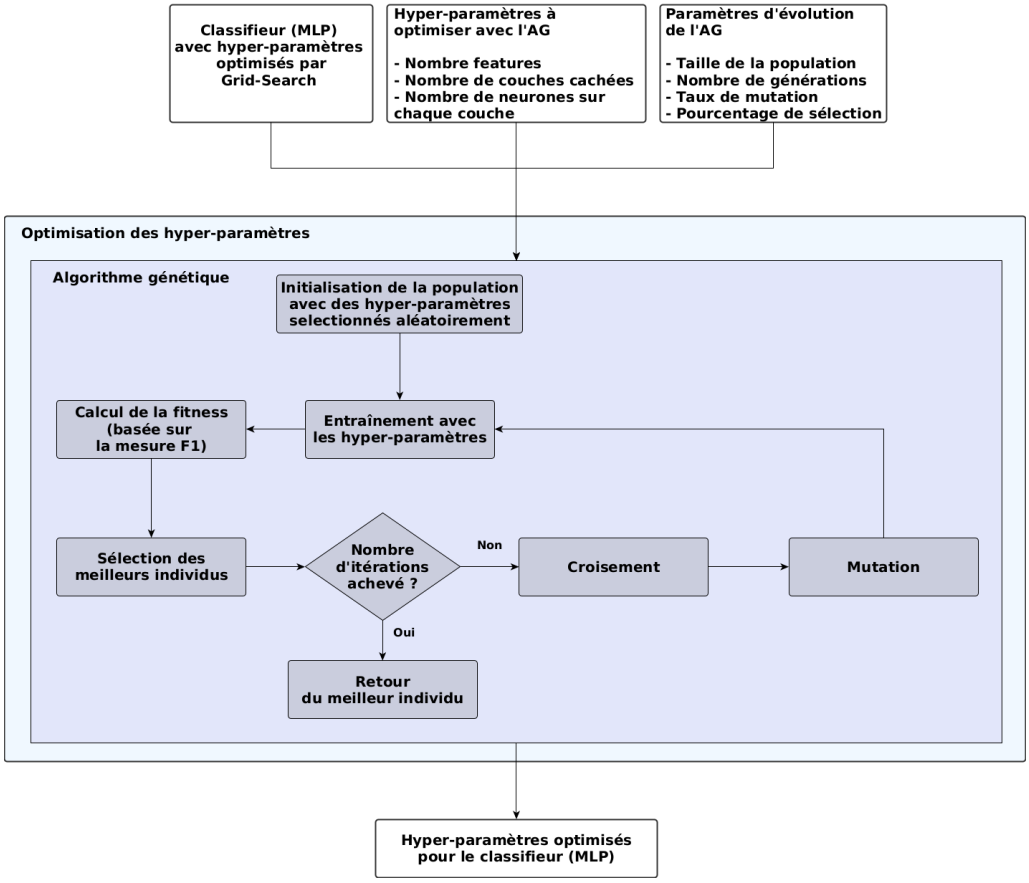



FIGURE II.4.4 – Processus de sélection du classifieur.

La Figure II.4.4 présente l'algorithme génétique que nous avons implémenté. Les AGs recherchent les solutions en utilisant des méthodes stochastiques inspirées de mécanismes biologiques comme les mutations, les croisements ou encore la sélection naturelle. Un ensemble initial de solutions,

appelé population, est généré aléatoirement. Chaque solution, appelée individu, est décrite par un ensemble de caractéristiques (liste de valeurs) nommées chromosomes. Chaque individu est évalué à l'aide d'une fonction objectif. Le score de chaque individu est appelé sa valeur sélective ou *fitness* [Dar09]. Comme dans la théorie de la sélection naturelle, les individus ayant la *fitness* la plus haute sont ceux qui ont les plus grandes chances de survie. Ils ont de ce fait également les plus grandes chances de se reproduire. Un sous-ensemble des meilleurs individus nommés parents sont sélectionnés pour générer un nouvel ensemble d'individus appelés enfants. Chaque individu enfant est issu de deux parents par croisement de leurs chromosomes. Des mutations aléatoires sont appliquées pour conserver de la diversité dans les populations générées. Ces mutations permettent de maintenir l'équilibre entre recherche dans l'espace des solutions et convergence vers une solutions potentiellement optimale. Les enfants remplacent les individus les moins adaptés (ayant une faible *fitness*) et créent une nouvelle génération. La taille de la population reste constante au fil des générations. Ce processus de génération, croisement, mutation et sélection est itéré jusqu'à ce qu'une condition d'arrêt soit atteinte. Cette condition est bien souvent un nombre maximum d'itérations.
La population initiale est générée en utilisant des valeurs d'hyper-paramètres choisies aléatoirement dans des intervalles que nous avons fixés :

- nombre de *features* ($[20000 - 60000]$). Nous utilisons cet intervalle car la borne inférieure (20000) est proche de la valeur empiriquement déterminée (voir Section 4.4.2.2) pour la sélection des classifieurs (30000). La borne supérieure est quant à elle proche des deux-tiers du nombre maximal de *features* (99351).
- nombre de couches cachées ($[2 - 15]$). La borne supérieure est choisie afin de limiter le temps de calcul. En effet, le nombre de couches ainsi que leurs tailles sont déterminants dans le temps d'entraînement. Ayant plusieurs dizaines d'individus à entraîner à chaque génération, nous fixons le nombre de couches maximal à 15 afin de rester dans un temps de calcul acceptable.
- la taille des couches cachées ($[1 - 30]$). Cette intervalle permet de la variabilité dans les structures de neurones générés. Nous avons la même contrainte que pour le nombre de couches cachées à savoir, rester dans un temps de calcul acceptable dans le cadre d'un AG.

De plus, l'algorithme génétique a ses propres paramètres qui contrôlent les aspects de la stratégie évolutionniste. Ces paramètres influent sur la taille de la population, les meilleurs individus à sélectionner ou encore la diversité à introduire à chaque génération :

- Taille de la population. Nous avons conduit différentes expériences avec des tailles de population de 50, 100, 200 et 300 individus. Jong et Spears [JS89] recommandant de choisir des tailles de populations modérées. Nous ne sommes pas allés au-delà de 300 afin de rester dans les recommandations de Jong et Spears.
- Nombre de générations. Ce nombre est fixé à 150 pour avoir un temps de calcul acceptable.
- Pourcentage des meilleurs individus à retenir dans la population ($p_{ret} = 20\%$).

- Probabilité de mutation pour chaque individu ($p_{mut} = 0.1$). La valeur $p_{mut}$ a été étudiée dans différents travaux [GRD97, RG+11, JS89] et a donc été fixée conformément à l'état de l'art.
- Probabilité de sélection aléatoire ($p_{sel} = 0.3$). Ce paramètre contrôle la probabilité de sélectionner aléatoirement un parent afin de maintenir la diversité dans la population [TG94].

Comme montré par la Figure II.4.4, l'algorithme génétique est composé de sept étapes :

1. La population est initialisée avec des individus générés aléatoirement. Chaque individu est composé de chromosomes représentant les valeurs des hyper-paramètres utilisés pour configurer le MLP. Ces chromosomes contiennent un nombre de *features* en entrée et un tuple représentant les couches cachées de neurones. Ainsi, une instance d'un individu $i$ peut-être décrite comme suit :

   $$i(27890, (11, 25, 18, 6, 9))$$

   27890 représente le nombre neurones sur la couche d'entrée avec 5 couches cachées contenant respectivement 11, 25, 18, 6 et 9 neurones de la deuxième à la dernière couche.

2. Le MLP correspondant à chaque individu est entraîné en utilisant une évaluation *split train/-test* 75/25 du jeu de données. Cette stratégie est utilisée ici pour sa rapidité et son moindre coût de calcul. Nous avons exclu une évaluation 10-*fold* car malgré sa fiabilité elle aurait été trop coûteuse en temps de calcul pour évaluer nos individus

3. La *fitness* de chaque individu correspond à la performance du MLP (mesure F1) sur la partie de test du jeu de données. La mesure F1 est choisie ici car elle requiert à la fois une bonne précision et un bon rappel.

4. Les individus sont classés en fonction de leur *fitness*. Les meilleurs individus sont ensuite sélectionnés avec le taux de sélection $p_{ret}$.

5. Les individus sélectionnés se reproduisent : leurs enfants sont générés grâce à un croisement des chromosomes des parents. Le couple de parents est aléatoirement choisi parmi les meilleurs individus. Le croisement est effectué à la fois sur le nombre de *features* et sur les couches cachées de neurones :

   - Le croisement du nombre de *features* est réalisé en utilisant la moyenne du nombre de *features* des deux parents.
   - Le croisement des couches de neurones est réalisé en utilisant un seul point de coupe au milieu des couches cachées des parents $a$ et $b$ [SA91]. Si le nombre de couches est impair, le point de coupe est arrondi à l'entier inférieur. Les enfants héritent ainsi de la moitié de la structure de leurs parents, comme montré dans l'exemple ci-dessous. La première moitié des couches du parent $a$ est transmise à l'enfant. Pour le parent $b$, la seconde moitié est transmise.

$$i_{parent_a}(21912, (12, 23, 8, 4))$$
$$\textit{Croisement}$$
$$i_{parent_b}(30023, (4, 23, 5, 13, 27))$$
$$\Downarrow$$
$$i_{enfant}(\lfloor(21912 + 30023)/2\rfloor, (12, 23, \cancel{8}, \cancel{4}) \cdot (\cancel{4}, \cancel{23}, 5, 13, 27))$$
$$\Downarrow$$
$$i_{enfant}(25967, (12, 23, 5, 13, 27))$$

6. Les mutations sont réalisées en changeant un gène aléatoirement dans un individu. Dans cette approche, les mutations influent seulement sur les couches cachées de neurones. Nous avons choisi de créer trois types de mutations choisies aléatoirement avec une probabilité égale ($= \frac{1}{3}p_{mut}$ chacune) :
   - Addition : ajoute une couche de neurones générée aléatoirement dans l'individu. Exemple :

$$i(21912, (12, 23, 45))$$
$$\textit{Addition}$$
$$\Downarrow$$
$$i(21912, (12, 23, 45, 18))$$

   - Délétion : supprime une couche de neurones aléatoirement dans l'individu. Exemple :

$$i(21912, (12, 23, 45))$$
$$\textit{Délétion}$$
$$\Downarrow$$
$$i(21912, (12, 45))$$

   - Substitution : choisit un nombre de neurones aléatoire dans l'intervalle autorisé du nombre de neurones par couche et sélectionne aléatoirement une couche dans l'individu pour en changer la taille. Exemple :

$$i(21912, (12, 23, 45))$$
$$\textit{Substitution}$$
$$\Downarrow$$
$$i(21912, (7, 23, 45))$$

7. Le meilleur individu trouvé est retourné quand le nombre fixé de générations est atteint. Cet individu contient les meilleures valeurs d'hyper-paramètres découvertes pour le MLP.

## 4.5 Résultats

### 4.5.1 Sélection du classifieur

Comme nous l'avons décrit en section Section 4.4.3, nous comparons six classifieurs différents en utilisant un protocole d'évaluation 10-*fold* et en comparant les valeurs des mesures F1 obtenues. Ces mesures sont faites sur les classifieurs *Stochastic Gradient Descent* (SGD), *Support Vector Machines* (SVMs), *Random Forest* (RF), *Ridge Regression* (RR), *k-Nearest Neighbors* (kNN) et *Multi-Layer Perceptron* (MLP). Les hyper-paramètres de chaque classifieur sont d'abord grossièrement optimisés en utilisant la méthode *Grid-Search* puis leurs performances sont mesurées. Les résultats de cette expérimentation sont présentés dans Table II.4.1. Ce tableau permet de visualiser les performances des différents classifieurs par le biais de leurs mesures F1 et des intervalles de confiance à 95% associés. Cet intervalle signifie qu'au moins 95% des valeurs sur les 10 *folds* se situent dans l'intervalle compris entre $[\text{Mesure F1} - \text{IC 95\%}, \text{Mesure F1} + \text{IC 95\%}]$
L'algorithme de *Grid-Search* a trouvé un ensemble d'hyper-paramètres optimisés pour les six classifieurs. Les couples classifieur/hyper-paramètres et les résultats sont visibles dans la Table II.4.1. Une fois les hyper-paramètres optimisés, le classifieur obtenant les meilleurs résultats est le MLP (0.868). C'est sur la base de ce résultat que nous avons sélectionné le MLP comme classifieurs à optimiser finement à l'aide d'un algorithme génétique.

| **classifieur** | **Hyper-paramètres optimisés par Grid-Search** | **Mesure F1** | **Protocole d'évaluation** |
|---|---|---|---|
| **MLP** | activation='tanh'<br>learning_rate='adaptive'<br>max_iter=100<br>random_state=0 | **0.868**<br>**IC 95%: 0.034** | 10-*fold* |
| SVM | C=100<br>gamma='scale' | 0.857<br>IC 95%: 0.042 | 10-*fold* |
| SGD | loss='modified_huber'<br>max_iter=5000<br>random_state=0 | 0.841<br>IC 95%: 0.037 | 10-*fold* |
| RR | random_state=0 | 0.819<br>IC 95%: 0.050 | 10-*fold* |
| RF | criterion='entropy'<br>n_estimators=20<br>random_state=0 | 0.610<br>IC 95%: 0.089 | 10-*fold* |
| kNN | weights='distance'<br>n_neighbors=2 | 0.449<br>IC 95%: 0.107 | 10-*fold* |

TABLE II.4.1 – Résultats obtenus avec les six classifieurs testés avec 30000 *features* et une validation 10-*fold*.

### 4.5.2 Résultats intermédiaires

Nous avons évalué l'influence de chaque phase (TALN, sélection des variables représentatives, etc.) du processus de classification sur la performance. Pour cela nous avons créé cinq configurations différentes sur lesquelles nous avons calculés quatre métriques : précision, rappel, mesure F1 et justesse. Les résultats de ces expérimentations sont montrés par la Figure II.4.5.

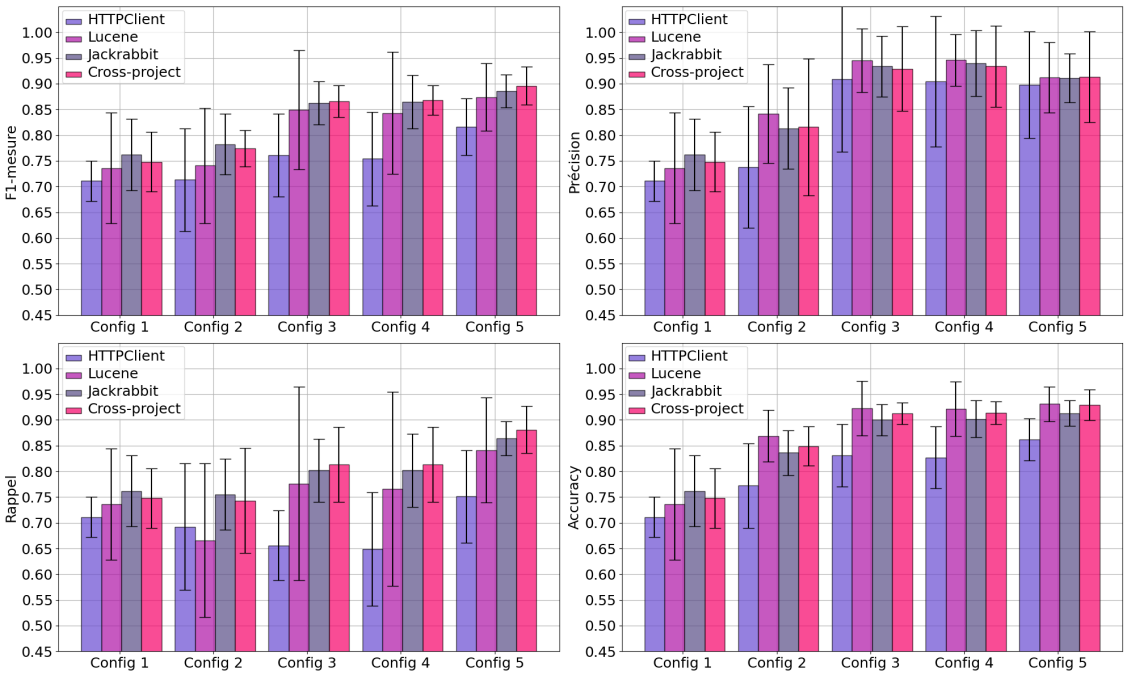



FIGURE II.4.5 – Processus de sélection du classifieur.

La configuration 1 utilise uniquement des uni-grammes et TF-IDF sans atténuation logarithmique de la fréquence des termes. Dans cette configuration aucune sélection de *features* n'est effectuée et le MLP utilise les valeurs par défaut des hyper-paramètres tels que définis par l'API Scikit-learn.
Dans la configuration numéro 2, TF-IDF utilise des uni-grammes et bi-grammes ainsi qu'une atténuation logarithmique de la fréquence des termes. De plus, les mots sur et sous-représentés sont filtrés. Ici l'ensemble des *features* est utilisé (63924). Les paramètres du MLP sont toujours ceux définis par défaut. Un gain notable en termes de précision (+0.068) et de justesse (+0.101) est mesuré inter-projets par rapport à la configuration 1. L'augmentation de la mesure F1 reste faible (+0.027) à cause d'une valeur de rappel peu élevée.
La configuration 3 utilise TF-IDF avec des uni-grammes, bi-grammes et tri-grammes. Dans cette configuration nous utilisons également une sélection des *features* les plus représentatives par un test du chi-deux (30000 *features*). Les hyper-paramètres du MLP sont encore gardés à leurs valeurs par défaut. L'utilisation des tri-grammes et de la sélection de *features* a une influence positive par rapport à la configuration 2 sur le rappel (+0.070). Cette augmentation du rappel influence la

mesure F1 positivement (+0.092).
Dans la configuration 4 nous utilisons la même vectorisation avec TF-IDF et la sélection de *features* que dans la configuration 3. Nous utilisons un MLP avec les hyper-paramètres trouvés par GridSearch. Malgré l'utilisation d'une méthode d'optimisation, le gain sur l'ensemble des mesures est presque nul.
La dernière configuration (configuration 5) utilise les mêmes configurations de TF-IDF et de sélection de *features* que les configurations 3 et 4. À la différence de la configuration 4, nous utilisons les hyper-paramètres optimisés par l'algorithme génétique pour le classifieur MLP. Sur l'ensemble des projets, mais également sur la mesure inter-projets, le rappel se trouve amélioré (+0.068) par rapport à la configuration 4. La précision est légèrement dégradée mais reste stable sur l'ensemble des projets et spécifiquement sur la mesure inter-projets. En utilisant une optimisation par AG, la mesure F1 est améliorée pour l'ensemble des projets et sur la mesure inter-projets (+0.061). La classification est ainsi légèrement moins précise mais plus robuste.
Ces configurations montrent l'impact globalement positif des différentes étapes et paramètres utilisés dans notre approche. Nous obtenons une configuration utilisable (configuration 5) ayant des performances relativement stables sur l'ensemble des mesures effectuées. Les résultats de l'algorithme génétique et la comparaison avec l'état de l'art sont décrits dans la section suivante.

### 4.5.3 Optimisation par algorithme génétique du classifieur

Dans la section précédente, nous avons détaillé les résultats des différentes configurations. Dans cette section, nous allons décrire les résultats de l'algorithme génétique mais aussi la comparaison avec l'état de l'art. Dans la Section 4.4.4, nous avons décrit la méthode d'optimisation du MLP à l'aide d'un algorithme génétique. L'algorithme génétique est exécuté avec les paramètres définis dans la Section 4.4.4 sur 150 générations. La Figure II.4.6 montre l'évolution du score de *fitness* au fil des générations avec quatre tailles de population différentes : 50, 100, 200 et 300 individus. Les plus petites tailles de population avec 50 et 100 individus obtiennent les meilleurs résultats tandis que les populations plus grandes obtiennent des résultats moindres. Ces différences peuvent s'expliquer à cause du mécanisme de sélection des parents que nous avons implémenté. Les parents étant aléatoirement sélectionnés dans un sous-ensemble de la population en fonction d'une proportion fixe de meilleurs individus (20%), une population plus petite implique un nombre de parents plus petit. Cela peut amener à une stratégie d'évolution plus élitiste conduisant à une amélioration plus rapide du score de *fitness*. L'exécution de l'algorithme génétique avec 50 individus et 150 générations renvoie un individu optimisé avec une configuration qui utilise 7 couches cachées avec des tailles variant de 9 à 15 et 37 362 *features* en entrée : $i(37362, (15, 9, 10, 11, 9, 15, 11))$.
Nous comparons ensuite les performances du meilleur individu retourné par l'algorithme génétique aux résultats de l'état de l'art. Pour cela, nous validons cet individu avec un 10-*fold* sur le jeu de données de 5591 tickets. Nos résultats sont présentés en bleu dans la Table **??** en comparaison avec ceux obtenus par Terdchanakul et al. [THPM17] en gris (meilleurs résultats de l'état de l'art). L'approche que nous proposons obtient un gain de +0.016 sur la mesure F1 moyenne par rapport

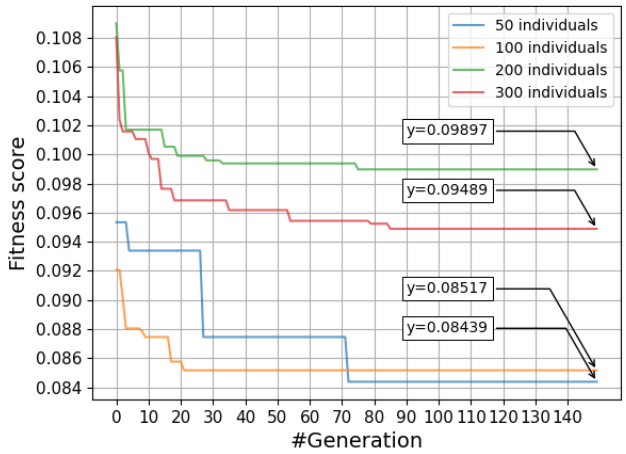



FIGURE II.4.6 – Évolution du *fitness* sur 4 tailles de populations différentes.

| **Mesure F1 (évaluation 10-*fold*)** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Terdchanakul *et al.* | | | | | | | | |
| HTTPClient | | Jackrabbit | | Lucene | | Moyenne sur 3 projets | inter-projet | |
| 0.814 | | 0.805 | | 0.884 | | **0.843** | **0.814** | |
| MLP optimisé avec algorithme génétique | | | | | | | | |
| HTTPClient | | Jackrabbit | | Lucene | | Moyenne sur 3 projets | inter-projet | |
| 0.816 (+0.002) | CI 95%: 0.055 | 0.886 (+0.081) | CI 95%: 0.032 | 0.874 (-0.010) | CI 95%: 0.066 | **0.859 (+0.016)** | **0.896 (+0.082)** | CI 95%: 0.037 |

TABLE II.4.2 – Comparaison entre les résultats obtenus par Terdchanakul *et al.* [THPM17] (en gris) et notre approche (en bleu).

aux résultats obtenus par Terdchanakul et al. sur l'ensemble des projets. Notre solution améliore les résultats pour deux projets sur trois : HTTPClient (+0.002) et Jackrabbit (+0.081). Une faible détérioration est observée sur le projet Lucene (-0.010). La mesure F1 sur la mesure inter-projets subit elle une forte amélioration (+0.081) passant de 0.814 à presque 0.9. Comme montré dans la Figure II.4.5, notre solution atteint 0.881 en rappel, 0.913 en précision et 0.929 en *accuracy* sur la mesure inter-projets.

## 4.6 Application du classifieur à d'autres cas d'études

Dans la Section 4.5.3, nous avons montré que le classifieur MLP avait de bons résultats sur le jeu de données de Herzig *et al.* [HJZ13] en utilisant une validation 10-*fold*. Le classifieur ayant de bonnes performances en contexte expérimental, nous avons souhaité le confronter à des tickets inconnus issus d'autres projets *open-source*. Pour cela, nous avons extrait des ensembles de tickets de 3 autres projets provenant de GitHub.

Le premier projet est *BroadleafCommerce* (décrit en section Section 5.4.1). *BroadleafCommerce* est un bon candidat car c'est un projet *open-source*, écrit en Java, qui contient un grand nombre de tickets annotés par les développeurs.
Le deuxième projet est Apache Pulsar, un système distribué de messagerie du type *publish-subscribe*. Initialement développé par l'entreprise Yahoo!, il a été mis en *open-source* en 2016 par la fondation Apache. Apache Pulsar dispose d'un grand nombre de *stars* (environ 8500) et de *forks* (plus de 2100) ce qui suggère un bon support de recherche selon Kalliamvakou *et al.* [KGB+16].
Le troisième projet auquel nous confrontons notre classifieur est JHipster. Celui-ci est un générateur d'applications web permettant d'obtenir deux piles technologiques (cliente et serveur) utilisables très rapidement. JHipster est majoritairement écrit en Javascript et Typescript mais utilise un vaste panel de technologies pour la génération d'applications. C'est un projet populaire sur GitHub (environ 18400 *stars* et 3500 *forks*). JHipster a également été le support de plusieurs travaux de recherche concernant les lignes de produits et les tests de configurations [HNA+17, HNA+19]. Dans notre cas, nous souhaitons confronter notre classifieur à la diversité des technologies de JHipster.

### 4.6.1 Étiquetage des jeux de tickets

Nous avons utilisé l'API HTTP GitHub pour extraire l'ensemble des tickets des trois projets. Comme dans le jeu de données de Herzig *et al.*, les tickets issus de ces projets ont une structure composée d'un titre et d'une description. Nous avons ensuite catégorisé les tickets en fonction de leurs étiquettes :

- "BUG" pour les tickets étant étiquetés par les développeurs comme tels,
- "NBUG" pour les tickets n'ayant pas de catégories ou ceux ayant été étiquetés par les développeurs avec une autre étiquette que "bug".

Les résultats de cet étiquetage sont montrés sur la Table II.4.3.

### 4.6.2 Résultats de classification

Nous avons entraîné le MLP sur les 5591 tickets du jeu de données de Herzig *et al.* [HJZ13]. L'entraînement effectué, nous avons utilisé notre modèle pour prédire les étiquettes des tickets sur les 3 projets. La Table II.4.3 montre les résultats de classification obtenus. Le classifieur obtient d'assez bons résultats sur des projets (*BroadleafCommerce*, Apache Pulsar) utilisant le même langage (Java) que les projets utilisés pour l'entraînement. Notons que les jeux de tickets récupérés ici ont été annotés par les développeurs et n'ont pas été nettoyés manuellement. Ce faisant, il est possible que des erreurs de classifications par les développeurs existent. Notre classifieur a de bonnes performances sur *BroadleafCommerce* : 0.6980 en précision, 0.5272 en rappel et 0.6007 en mesure F1. Sur le projet Apache Pulsar le classifieur a des performances moyennes mais qui cependant restent acceptables : 0.6058 en précision, 0.4725 en rappel et 0.5309 en mesure F1. En revanche, le MLP a beaucoup de difficultés sur un projet utilisant des technologies autres que Java. Sur le

| | #tickets BUG | #tickets NBUG | Précision | Rappel | Mesure F1 | Justesse |
|---|---|---|---|---|---|---|
| **BroadleafCommerce** | 1140 | 1369 | 0.6980 | 0.5272 | 0.6007 | 0.6815 |
| **Apache Pulsar** | 2950 | 7715 | 0.6058 | 0.4725 | 0.5309 | 0.7691 |
| **JHipster** | 836 | 14176 | 0.1058 | 0.3325 | 0.1606 | 0.8064 |

TABLE II.4.3 – Résultats des mesures de classification sur les tickets des projets, *BroadleafCommerce*, Apache Pulsar et JHipster.

projet JHipster le classifieur obtient : 0.1058 en précision, 0.3325 en rappel et 0.1606 en mesure F1.

Nous avons ensuite expertisé le nombre de résultats vrais positifs, vrais négatifs, faux positifs et faux négatifs issus du classifieur MLP pour chacun des projets. Pour cela, nous avons produit les matrices de confusion (Figure II.4.7) permettant de visualiser ces différentes catégories. Nous pouvons observer qu'une majorité de tickets de *BroadleafCommerce* et Apache Pulsar sont classés comme des faux positifs (539 et 907), une plus faible part sont classés (260 et 907) comme des faux négatifs. Ces erreurs de classification ont trois explications. La première provient de la qualité de notre classifieur qui malgré ses bons résultats n'est pas parfait. La seconde est liée à un mauvaise étiquetage des tickets par les développeurs. Enfin, il est probable que les développeurs de *BroadleafCommerce* et d'Apache Pulsar ne rédigent pas leurs tickets comme dans les trois projets ayant servi pour l'entraînement.

### 4.6.3 Explicabilité de la classification des tickets

Précédemment, nous avons commenté les résultats de classification sur *BroadleafCommerce* en nous focalisant sur les métriques de performances. Dans cette section nous observons et commentons l'explicabilité de la classification des tickets au travers d'exemples représentatifs qui permettent de visualiser les *features* qui pèsent le plus dans la classification. L'explicabilité d'une classification peut s'obtenir de deux manières :

- par la nature et les paramètres ajustés du classifieur. La classification d'une instance donnée par un classifieur linéaire va s'expliquer à l'aide de la position de l'instance par rapport au plan de séparation défini par le classifieur. La classification d'une instance donnée par un arbre de décision peut s'expliquer par les feuilles de l'arbre à parcourir pour arriver au résultat de classification.
- en utilisant une méthode d'explicabilité post-entraînement (méthode *post-hoc*). Ces méthodes s'appliquent généralement à des classifieurs "boîtes noires" dont l'explicabilité est rendue compliquée par nature. En effet, des modèles simples tels que les modèles linéaires permettent une explication relativement aisée de la classification des instances. Or, dans le cas de modèles plus complexes comme les réseaux de neurones utilisés ici, l'explicabilité n'est pas facilement appréhendable. Parmi les méthodes permettant d'expliquer la classification d'instances textuelles deux types coexistent :

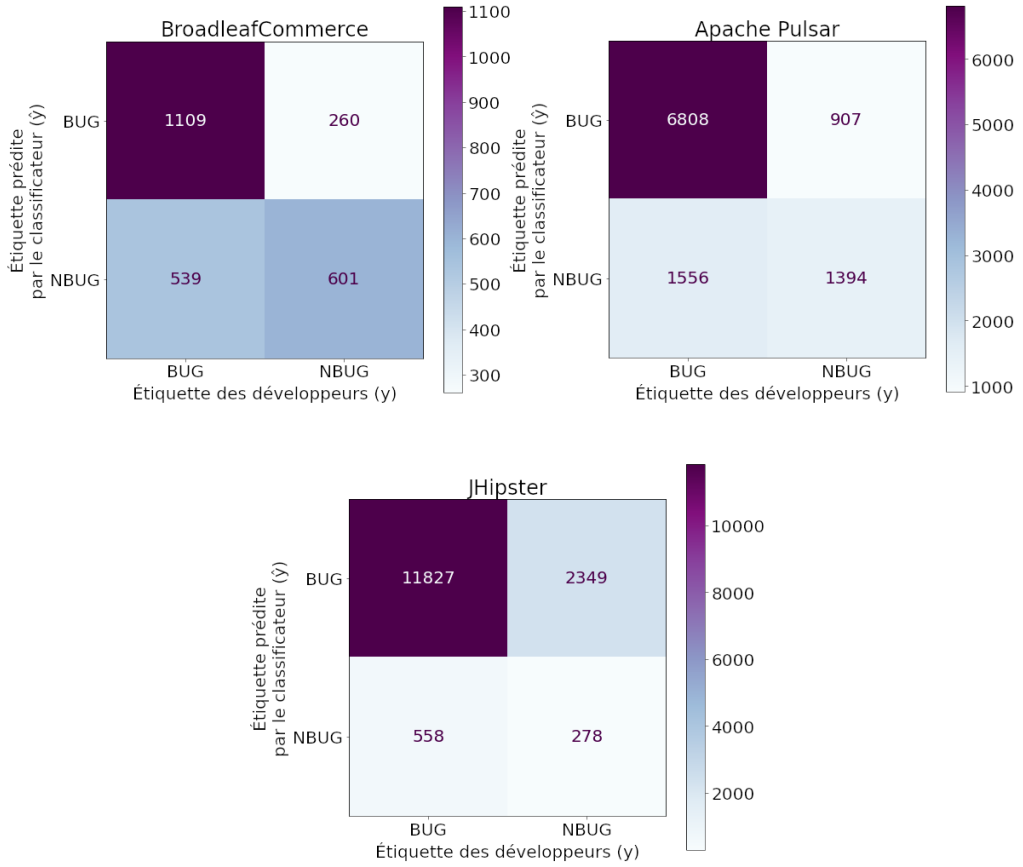



FIGURE II.4.7 – Matrices de confusion issues de la classification des tickets sur les projets : *BroadleafCommerce*, Apache Pulsar et JHipster.

- les méthodes nécessitant des modèles de langues afin d'expliquer la prédiction du classifieur, telles que Anchor [RSG18] ou *SHapley Additive exPlanations* (SHAP) [LL17].
- la méthode *Local Interpretable Model-agnostic Explanations* (LIME) [RSG16]. LIME n'a besoin d'aucun modèle externe particulier pour expliquer une prédiction (modèle agnostique). C'est pourquoi nous choisissons LIME pour expliquer les résultats de classification de notre MLP.

Le principe de fonctionnement de LIME est relativement simple. Il transforme un modèle complexe en un modèle linéaire facilement interprétable en s'appuyant sur les données au voisinage de l'instance à expliquer. LIME présente les instances classifiées en surlignant d'une manière spécifique les mots orientant la classification avec un gradient de couleurs. Un mot surligné en vert influe positivement sur la catégorie prédite pour un ticket. Un mot surligné en rouge influe négativement sur la catégorie prédite.

#### 4.6.3.1 Faux négatifs

Les premières interprétations que nous faisons concernent trois tickets produisant des faux négatifs. Nous avons manuellement examiné les faux négatifs puis sélectionné un candidat pertinent dans les trois projets.

Le ticket de *BroadleafCommerce* de la Figure II.4.8 produit un faux négatif relativement révélateur d'une erreur de classification par les développeurs mais d'une bonne classification par le MLP. En effet, ce ticket décrit explicitement un *refactoring*. On peut également observer que les mots "performances", "refactored" ou encore "inefficient" ont un impact positif sur la classification en tant que non-bug. Ces termes sont souvent utilisés pour décrire un demande d'amélioration d'une fonctionnalité. Il est donc pertinent que le classifieur les considère comme des termes orientant vers la catégorie "non-bug".

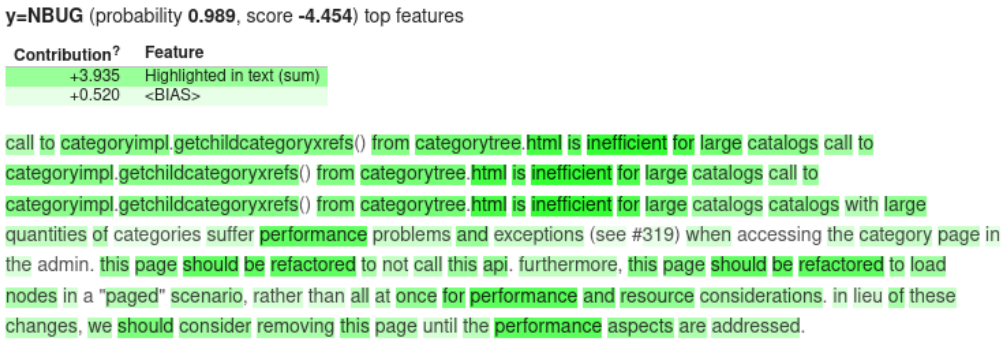
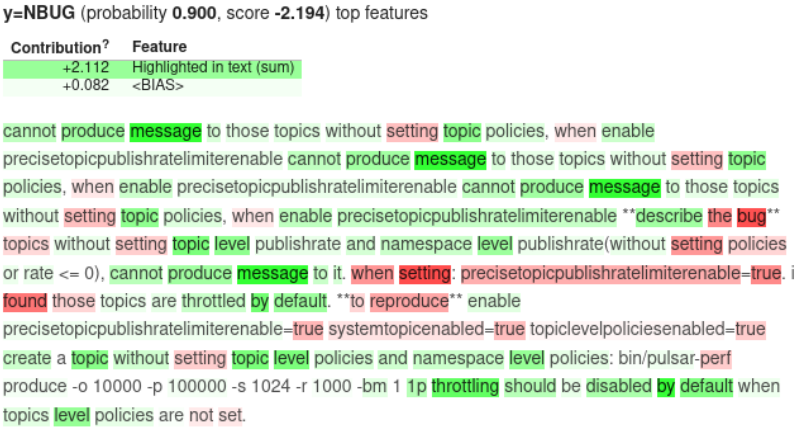



FIGURE II.4.8 – Ticket de *BroadleafCommerce* considéré comme faux négatif.

Le ticket d'Apache Pulsar en Figure II.4.9 crée un faux négatif issu d'une erreur de classification du MLP. De manière assez surprenante, les termes tels que "message", "produce", "level" ou encore "throttling" influencent la prédiction vers la classe "non-bug". En revanche, les termes "setting", "bug" ou "found" orientent la prédiction vers la classe "bug". Ici, la multiplication du titre par 3 joue défavorablement sur le classifieur. Les termes "produce" et "message" orientant la classification vers la classe "non-bug", leur multiplication influe fortement sur la décision du classifieur.



**y=NBUG** (probability **0.900**, score **-2.194**) top features

| Contribution? | Feature |
|---|---|
| +2.112 | Highlighted in text (sum) |
| +0.082 | <BIAS> |

cannot produce message to those topics without setting topic policies, when enable precisetopicpublishratelimiterenable cannot produce message to those topics without setting topic policies, when enable precisetopicpublishratelimiterenable cannot produce message to those topics without setting topic policies, when enable precisetopicpublishratelimiterenable **describe the bug** topics without setting topic level publishrate and namespace level publishrate(without setting policies or rate <= 0), cannot produce message to it. when setting: precisetopicpublishratelimiterenable=true. i found those topics are throttled by default. **to reproduce** enable precisetopicpublishratelimiterenable=true systemtopicenabled=true topiclevelpoliciesenabled=true create a topic without setting topic level policies and namespace level policies: bin/pulsar-perf produce -o 10000 -p 100000 -s 1024 -r 1000 -bm 1 1p throttling should be disabled by default when topics level policies are not set.

FIGURE II.4.9 – Ticket d'Apache Pulsar considéré comme faux négatif.

Le ticket présenté en Figure II.4.10 produit un faux négatif issu d'une erreur de classification des développeurs et d'une bonne classification par le MLP. Ce ticket décrit un besoin de mise à jour sur une implémentation. On observe que les termes reliés à ce besoin tels que "upgrade", "implementation", "migrate" ou "deprecated" orientent nettement la classification sur "non-bug".

**y=NBUG** (probability **0.985**, score **-4.167**) top features

| Contribution? | Feature |
|---|---|
| +3.485 | Highlighted in text (sum) |
| +0.682 | <BIAS> |

upgrade r2dbc implementation for spring boot 2.4 upgrade r2dbc implementation for spring boot 2.4 upgrade r2dbc implementation for spring boot 2.4 ##### **overview of the issue** from the [spring boot 2.4 release notes](https://github.com/spring-projects/spring-boot/wiki/spring-boot-2.4-release-notes): > the core infrastructure of r2dbc has moved to spring framework with a new `spring-r2dbc` module. if you are using this infrastructure, make sure to migrate deprecated access to the new core support. i'm creating this as a separate issue than the [main spring boot 2.4 issue](https://github.com/jhipster/generator-jhipster/issues/13237) because it requires more work than expected and i want to add a bug bounty to it. ##### **motivation for or use case** our current implementation of r2dbc does not work with spring boot 2.4. [this pr](https://github.com/jhipster/generator-jhipster/pull/13551) is tracking spring boot 2.4 support and apps with r2dbc do not start successfully because the expected `databaseclient` is not available. ##### **reproduce the error** `npm link` in the `spring-boot-2.4` branch, then create an app with the following `.yo-rc.json` and try to start it. ```json { "generator-jhipster": { "applicationtype": "monolith", "reactive": true, "basename": "samplewebfluxpsql", "packagename": "tech.jhipster.sample", "packagefolder": "tech/jhipster/sample", "authenticationtype": "jwt", "cacheprovider": "no", "enablehibernatecache": false, "websocket": false, "databasetype": "sql", "devdatabasetype": "h2disk", "proddatabasetype": "postgresql", "searchengine": false, "buildtool": "maven", "enabletranslation": true, "nativelanguage": "en", "languages": ["en", "fr"], "testframeworks": ["gatling", "cypress"], "serverport": "8080", "jhiprefix": "jhi", "clientpackagemanager": "npm", "enableswaggercodegen": true, "clientframework": "angularx", "creationtimestamp": 1596513172471, "jwtsecretkey": "zjy4mtm4yji5yzmwzjhjyji2otnkntrjmwq5y2q0y2ywownmzte2nzrmyzu3ntmwm2njote3mtllotm3mwrkn "blueprints": [], "othermodules": [], "jhipsterversion": "7.0.0-beta.1", "skipclient": false, "skipserver": false, "skipusermanagement": false, "skipchecklengthofidentifier": false, "skipfakedata": false, "entitysuffix": "", "dtosuffix": "dto", "pages": [], "servicediscoverytype": false, "messagebroker": false, "clienttheme": "none", "clientthemevariant": "", "withadminui": true } } ``` ##### **suggest a fix** upgrade to non-deprecated classes. ##### **jhipster version(s)** `spring-boot-2.4` branch

FIGURE II.4.10 – Ticket de JHipster considéré comme faux négatif.

#### 4.6.3.2 Faux positifs

Tout comme nous l'avons fait dans la section précédente, nous allons examiner un faux positif pour chacun des trois projets étudiés.
Le ticket de *BroadleafCommerce* de la Figure II.4.11 résulte en un faux positif issu d'une erreur de classification par les développeurs mais d'une bonne classification par le MLP. Ce ticket est relativement ambigu pour le classifieur. Cela s'observe par une grand nombre de termes comme : "anonymous", " clean" ayant une influence négative sur la classification "bug". De manière surprenante, le morceau de phrase "many thanks for support" oriente également la classification vers la classe "non-bug". Un explication plausible est que les développeurs utilisent cette formule de politesse dans des tickets ne décrivant pas des bogues. Les termes "following", "exception" ou "fail" orientent la classification positivement vers la classe "bug".

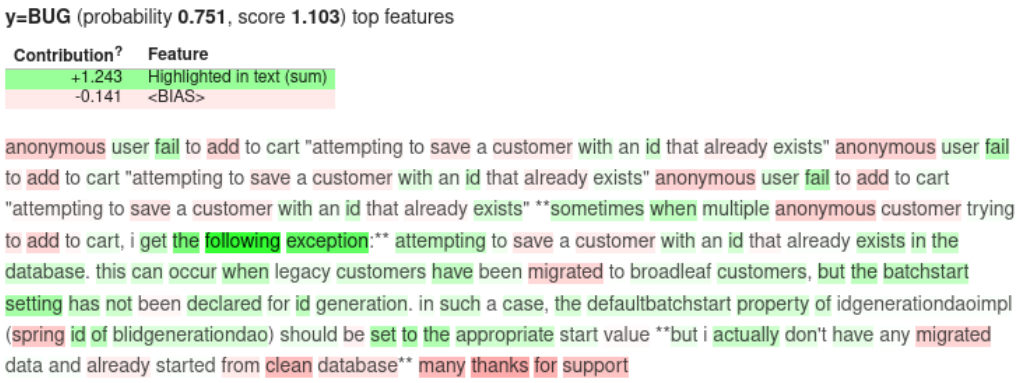



FIGURE II.4.11 – Ticket de *BroadleafCommerce* considéré comme faux positif.

La Figure II.4.12 montre un ticket issu d'Apache Pulsar créant un faux positif par erreur de classification du MLP. Le ticket de la Figure II.4.12 décrit un besoin de documentation et non un bug. Ici peu de termes ont un poids important sur la classification. Cela montre une certaine ambiguïté sur la rédaction du ticket d'autant plus que la description de celui-ci est courte.

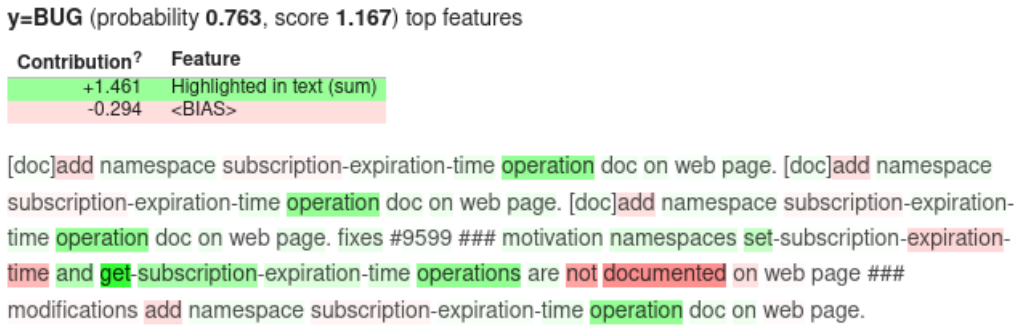



FIGURE II.4.12 – Ticket d'Apache Pulsar considéré comme faux positif.

Enfin, le faux positif de la Figure II.4.13 correspond à un ticket issu de JHipster décrivant un bogue. Ce ticket est mal étiqueté par les développeurs mais correctement classifié par le MLP. Ici on observe que des mots comme "forbidden", "error", "token" ou "403" orientent fortement la classification vers la classe "bug".

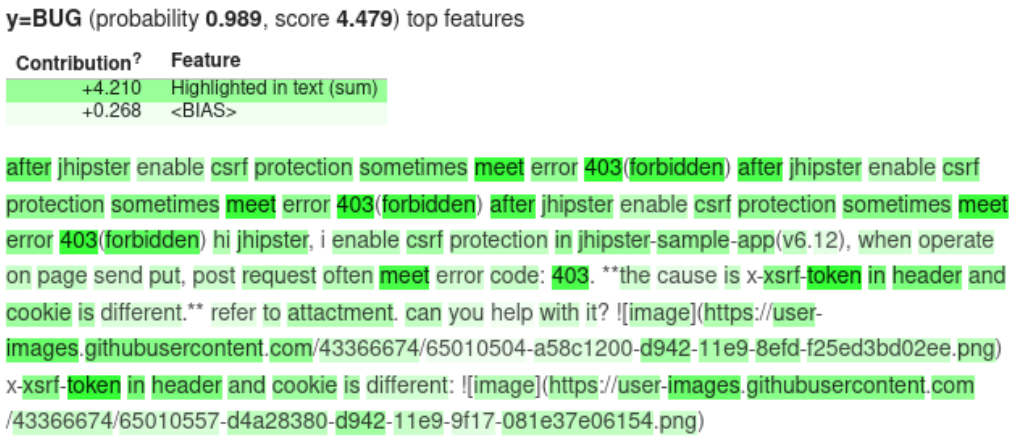



FIGURE II.4.13 – Ticket de JHipster considéré comme faux positif.

## 4.7 Prototype d'assistance à la rédaction de tickets

Une perspective intéressante de ces travaux concerne l'assistance à la rédaction de tickets. Nous avons suivi cette perspective et créé un prototype qui permet d'assister les développeurs dans la rédaction de leurs tickets de bogues. Ce prototype utilise le classifieur présenté en Section 4.4. L'idée de ce prototype est de fournir aux développeurs une interface leur permettant de rédiger un ticket contenant un titre et une description puis d'utiliser le classifieur pour prédire la catégorie de ce ticket. Ce prototype se présente sous la forme d'une interface web, illustrée par la Figure II.4.14 et disponible en ligne [2].

Dans cette interface, nous trouvons deux champs de texte permettant de saisir les informations composant le ticket. La fenêtre à gauche "Probable Next Word" est une interface de suggestion de mots pour remplir les champs du tickets. Cette fonctionnalité est basée sur des chaînes de Markov. Ces chaînes permettent d'estimer la probabilité d'apparition d'un mot en fonction du ou des mots précédents. Nous avons créé des chaînes de Markov permettant la suggestion d'un mot sachant un mot précédent, mais également sachant deux et également trois mots précédents, ce qui permet une suggestion plus pertinente des mots. Nous fournissons à l'utilisateur la probabilité du mot suggéré dans l'interface. Nous avons également ajouté une fonctionnalité d'auto-complétion par les mots suggérés dans les champs textuels.

Une fois le ticket rédigé, l'utilisateur a la possibilité de prédire le type de ticket. L'application lui retourne alors un message l'informant si le ticket rédigé décrit un bogue ou non.

2. http://ec2co-ecsel-1iegdism3qjis-2023048106.eu-west-3.elb.amazonaws.com/

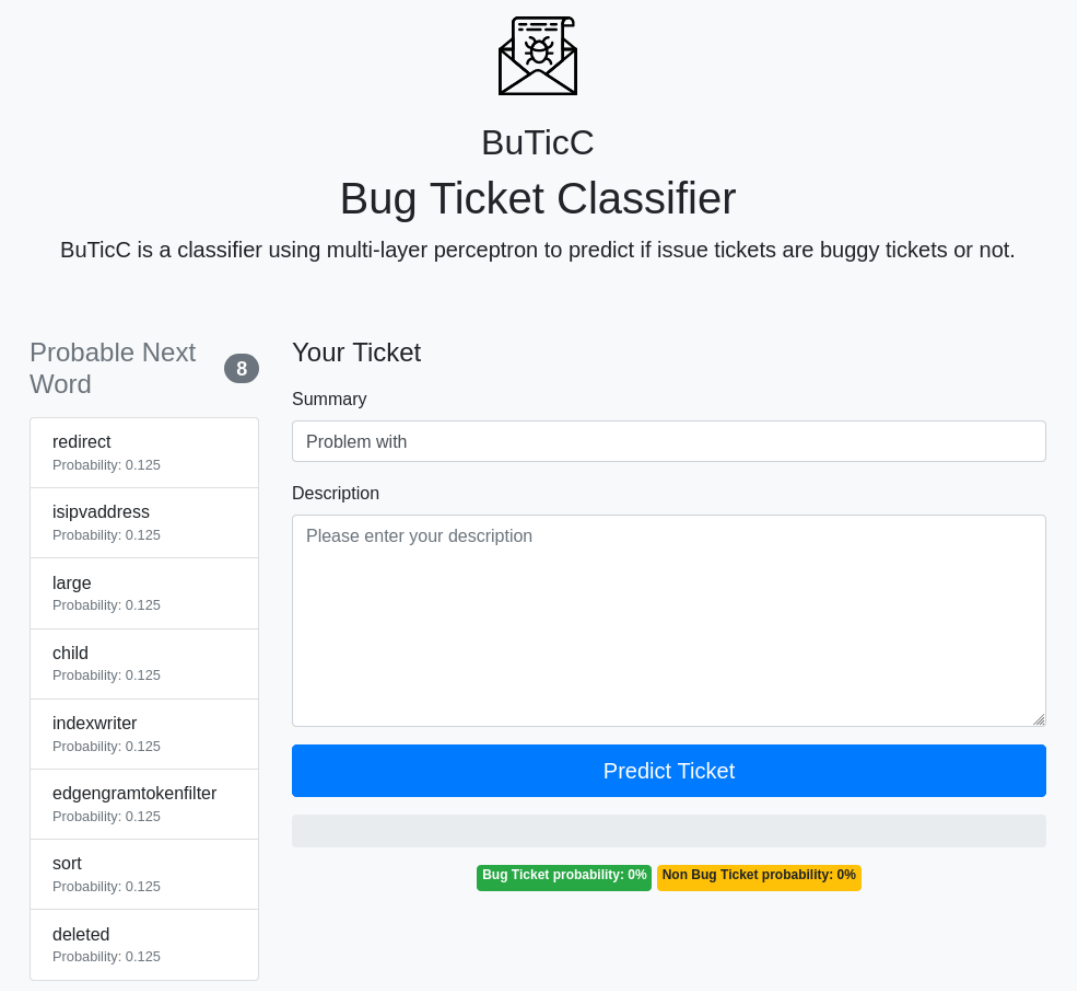



FIGURE II.4.14 – Prototype d'assistance à la rédaction de tickets, suggestion de mots par chaînes de Markov et évaluation dynamique de la probabilité d'appartenir à la classe bogue.

## 4.8 Synthèse, limites et perspectives

Dans ce chapitre, nous avons créé une approche de classification des tickets de bogues. Cette approche utilise une phase de traitement automatique du langage naturel dans laquelle nous supprimons les mots sur-représentés et sous-représentés. Nous transformons ensuite ce corpus en vecteurs à l'aide de la méthode TF-IDF. Puis, nous sélectionnons les variables les plus représentatives à l'aide de la méthode du chi-deux. Nous avons ensuite optimisé un panel de six classifieurs à l'aide d'une méthode combinatoire et comparé les performances de chacun. Puis, nous avons sélectionné celui ayant les meilleures performances à savoir un réseau de neurones multi-couches (MLP). Enfin, nous avons optimisé les paramètres de ce classifieur à l'aide d'un algorithme génétique.

### 4.8.1 Synthèse

**Synthèse Question de recherche 4.1**

L'approche développée dans ce chapitre (détaillée en Section 4.4) et les résultats obtenus permettent de répondre favorablement à la Question de recherche 4.1. Nous obtenons des résultats supérieurs à l'état de l'art. Sur la mesure F1 inter-projet, nous obtenons des performances de classifications supérieures (0.896) à celles de Terdchanakul *et al.* (0.814) [THPM17]. En plus d'avoir une haute valeur de mesure F1, nous obtenons un bon compromis entre rappel (0.881) et précision (0.913). L'ensemble de ces résultats montre que nous sommes capable de correctement cibler les tickets décrivant des bogues à l'aide de notre classifieur.

**Synthèse Question de recherche 4.2**

Afin de répondre à la Question de recherche 4.2, en Section 4.6, nous avons confronté notre classifieur à trois jeux de tickets inconnus provenant de trois projets *open-source* issus de GitHub. Les résultats que nous obtenons montrent que notre classifieur obtient de bons résultats de mesure F1 sur deux projets ayant le même contexte technologique (langage Java) que le jeu de données d'entraînement. En revanche, pour le projet utilisant un langage différent (TypeScript), les performances diminuent fortement. Cela montre que notre classifieur s'est trop spécialisé sur les technologies utilisées pour l'entraînement et qu'il ne peut donc pas être généralisé à d'autres langages ou technologies que celle utilisées dans le jeu d'entraînement. Nous avons également évalué les termes qui influencent la classification en Section 4.6.3. Nous avons vu que les termes ayant à trait à la maintenance orientent fortement la classification vers la catégorie non bogue. Nous avons vu également que la multiplication du titre du ticket pouvait avoir une forte influence sur la classification du ticket. Dans un ticket, un titre mal choisi pourrait influer plus fortement sur la classification que son contenu.

### 4.8.2 Incertitudes sur la validité

#### 4.8.2.1 Validité externe

Concernant la validité externe, notre classifieur a été construit sur un jeu de données contenant 5591 tickets issus de trois grands projets *open-source* écrits en Java. Le langage de programmation unique utilisé dans le jeu de données peut restreindre la généricité du classifieur. En effet, il se peut que les tickets contiennent des spécificités textuelles (morceaux de code Java, termes spécifiques) mettant en difficulté le classifieur sur d'autres langages. Une autre incertitude concernant la validité externe vient du fait que le jeu de données est composé exclusivement de tickets issus de projets *open-source*. Il est possible que des tickets issus de projets industriels non *open-source* soient rédigés

d'une manière différente du fait de contraintes ou de conventions liées à un contexte industriel ou une culture d'entreprise spécifique.

#### 4.8.2.2 Validité interne

Bien que les tickets aient été étiquetés manuellement par Herzig *et al.* [HJZ13], nous ne pouvons exclure de mauvais étiquetages résiduels qui viendraient fausser les résultats obtenus. Nous faisons également une sélection des *features* les plus représentatives du corpus de tickets en se basant sur le test statistique du Chi-deux. Ce test ayant un risque $\alpha = 0.05$, il se peut que des *features* ayant été statistiquement mesurées comme représentatives ne le soient pas.

### 4.8.3 Limites

Malgré de bons résultats, notre classifieur de tickets montre ses limites. La première vient de son aspect techno-centré. En effet, le classifieur étant entraîné sur des projets Java, son utilisation sur des projets utilisant d'autres langages et technologies entraîne une baisse importante des performances. Cela pourrait s'expliquer par un sur-ajustement du modèle sur les données d'entraînement (jeu de données de Herzig *et al.* [HJZ13]). La seconde limitation vient du fait que nous ne sommes capables de traiter que deux catégories de tickets ("*bug*" ou "*non-bug*").

### 4.8.4 Perspectives

Ces travaux ouvrent différentes perspectives.
La première est d'augmenter le nombre de catégories de tickets utilisables avec le classifieur afin de pouvoir utiliser le modèle dans un environnement de production de logiciels.
Une deuxième perspective pourrait être de spécialiser le classifieur sur des tickets de bogues liés à l'architecture logicielle. Un classifieur spécialisé permettrait d'alimenter des indicateurs de qualité logicielle dédiés à l'architecture. Il permettrait également une identification des développeurs ayant une compétence sur l'architecture logicielle ou une orientation des tickets concernant l'architecture vers les développeurs compétents.
Enfin, une dernière perspective concerne l'assistance à la rédaction de tickets par la classification automatique du ticket lors de sa rédaction. Grâce aux performances de notre classifieur, il est envisageable de l'utiliser comme une assistance pour détecter les tickets de bogues mal rédigés ou ambigus.

# Chapitre II.5

# Mesure de la contribution à l'architecture et catégorisation des développeurs

**Sommaire**

## 5.1 Introduction

Les métriques permettent aujourd'hui de détecter des *bad smells* [BFB99, FMZM16, ABT+18] mais aussi de mesurer la dette architecturale [RFPZ18]. D'autres métriques orientées processus permettent de mesurer la contribution et le *turn-over* des développeurs dans un projet [BNM+11, FFB14, FTL+15]. Cependant, à notre connaissance il n'existe pas de métrique spécifique permettant d'évaluer la contribution d'un développeur à l'architecture du logiciel.

Plusieurs études ont montré la faisabilité d'une mesure de la propriété de code source (*code ownership*) au travers des contributions faites par les développeurs au sein d'un projet. La mesure de propriété de code a été utilisée par Bird *et al.* [BNM+11] en 2011. Cette mesure permet la quantification de la contribution d'un développeur au sein d'un projet. Bird *et al.* ont utilisé la mesure de propriété avec une granularité définie au niveau des modules logiciels. Leurs travaux portent sur la mesure de propriété de code dans des modules de Windows Vista et Windows 7. Ils mesurent ensuite la relation avec le taux de bogues dans lesdits modules. Ces travaux mettent en exergue deux éléments majeurs : la conception d'une mesure de la propriété d'un code et la classification des développeurs en deux catégories : les développeurs dits *contributeurs majeurs* et les ceux dits *contributeurs mineurs*. La distinction entre ces deux catégories est basée sur un seuil de contribution au code d'un module. Bird *et al.* montrent également l'existence des contributeurs "héros", ayant un haut niveau de contribution pour un module spécifique mais qui contribuent moins à d'autres modules.

Foucault *et al.* [FFB14] répliquent l'étude de Bird *et al.* dans un contexte de sources ouvertes. En utilisant la mesure de la propriété et la catégorisation des développeurs [BNM+11], Foucault *et al.* [FFB14] étudient la propriété de code sur un corpus de projets *open-source*. Ils montrent qu'en contexte de sources ouvertes il n'y a pas de corrélation forte entre propriété du code et taux de faute dans les logiciels. Cela vient ainsi contredire les constations faites par Bird *et al.* [BNM+11] sur du code en sources fermées. Foucault *et al.* montrent également qu'en contexte *open-source* il n'y a pas de contributeur "héro" pour un seul module mais une multiplicité de développeurs avec un haut taux de contributions dans différents modules de l'application. Foucault *et al.* [FTL+15] étudient sept autres projets *open-source* afin de vérifier l'utilité d'une métrique sur la propriété d'un code source. Les conclusions de cette étude aboutissent aux mêmes constats que leur précédente étude [FFB14].

Dans ces études [BNM+11, FFB14, FTL+15], la mesure de propriété du code est réalisée à la granularité du module sans spécificités quant aux types de modifications (technologie spécifique, sémantique particulière, etc.) dénombrées au sein des modules. Une adaptation de cette métrique pour prendre en compte un travail sur des concepts ou des technologies spécifiques est donc envisageable.

L'objectif de ce chapitre est de montrer la faisabilité d'une métrique de propriété de code source appliquée à l'architecture logicielle et plus précisément à l'architecture *runtime* créée à l'aide du *framework* Spring. Pour cela nous formalisons une mesure de contribution globale à l'ensemble d'un projet logiciel mais également à l'architecture. Nous détaillons ensuite les analyses et les résultats

## 5.2 Modèle de contribution

Dans cette section nous commençons par définir un modèle formel de mesure de la contribution logicielle puis nous définissons des métriques en utilisant les éléments de ce modèle.

### 5.2.1 Formalisation d'une contribution logicielle

Les changements logiciels faits par différents développeurs sont les briques même du développement logiciel collaboratif. En considérant une version du projet et les versions précédentes, immédiates ou indirectes, la somme de ces changements peut être définie par un modèle de contribution. Ce modèle permet de lier chaque ligne modifiée dans chaque fichier avec l'auteur du changement. Il permet également d'identifier la nature, le contenu ainsi que le moment du changement. Ainsi, de manière plus formelle, nous définissons une contribution $c$ par le sextuplet suivant :

$$c(f, l, n, e, a, t)$$

avec :

- $f$ le fichier ayant subi des changements,
- $l$ la ligne de code modifiée dans le fichier $f$,
- $n$ le numéro de la ligne $l$ dans $f$,
- $e$ le type de changement sur la ligne $l$,
  - ADD pour un ligne ajoutée dans $f$,
  - DEL pour une ligne supprimée dans $f$,
  - MOD pour une ligne modifiée dans $f$,
- $d$ l'auteur (développeur) de la contribution,
- $t$ l'horodatage de la contribution.

```
Trinity 2017-06-14 17:55:46 94) @Bean
Neo     2017-10-16 10:33:55 95) public WebMvcRegistrations() {
Neo     2017-10-16 17:33:55 96)   return new WebMvcRegistrations(){
Trinity 2017-06-14 17:55:46 97)     @Override
Trinity 2017-06-14 17:55:46 98)     public RequestMappingHandlerMapping getRMHM() {
Trinity 2017-06-14 17:55:46 99)     return new AdminRequestMappingHandlerMapping();
Trinity 2017-06-14 17:55:46 100)    }
Trinity 2017-06-14 17:55:46 101)  };
Trinity 2017-06-14 17:55:46 101)}
```

Listing II.5.1 – Exemple de contributions de deux développeurs (*Trinity* et *Neo*) sur le fichier *AdminWebMvcConfiguration.java*.

Le Listing II.5.1 est un exemple réel extrait du projet *BroadleafCommerce*[1]. Cet exemple illustre les contributions de deux développeurs (pseudonymisés) sur le fichier *AdminWebMvcConfiguration.java*. Dans le Listing II.5.1, chaque ligne est une ligne du fichier *AdminWebMvcConfiguration.java* correspondant à une contribution. Par exemple, *Trinity* a ajouté sept lignes le 14/06/2017 à 17h55 tandis que *Neo* a modifié deux lignes le 16/10/2017 à 10h33.

### 5.2.2 Caractérisation du type de la contribution

Comme indiqué dans l'introduction de ce chapitre, nous étudions ici le code et les contributions qui permettent la gestion de l'architecture *runtime* d'un logiciel. Pour faciliter le développement et la maintenance des architectures *runtime*, les développeurs utilisent des *frameworks* proposant des mécanismes génériques d'instanciation de l'architecture *runtime*. Cela se traduit généralement par l'utilisation dans le code de marqueurs spécifiques aux technologies utilisées pour créer l'architecture *runtime*. Ainsi, connaissant les marqueurs associés à ces technologies, ceux-ci peuvent être utilisés pour caractériser les contributions. Ici, nous scindons les contributions au logiciel en deux catégories :
Pour faciliter le développement et la maintenance des architectures *runtime*, il est possible d'utiliser des *frameworks* proposant des mécanismes génériques d'instanciation de l'architecture *runtime*

- celles liées à l'architecture *runtime* appelées *contributions architecturales*,
- celles non liées à l'architecture *runtime* appelées *contributions non-architecturales*.

De manière formelle nous pouvons relier ces deux définitions au modèle précédemment défini en Section 5.2.1. Ainsi en considérant une fonction $words(l)$ de tokenisation d'une ligne de code $l$ nous définissons le type d'une contribution comme suit :

- $c(f, l, n, e, a, t)$ est une contribution,
- $M_a$ est l'ensemble des mots considérés comme marqueurs architecturaux d'une technologie,
- $r(c)$ définit le type de la contribution $c$ comme étant architectural (ARCH) ou non (DEV) :

$$r(c) = \begin{cases} \text{ARCH} & \text{iff} \exists\, w \in words(l) | w \in M_a \\ \text{DEV} & \text{else.} \end{cases} \tag{II.5.1}$$

### 5.2.3 Mesure de l'importance de la contribution par les auteurs

Nous avons bâti, en Sections 5.2.1 et 5.2.2, les éléments formels permettant de caractériser la contribution et de préciser son type. Comme vu dans l'état de l'art et en introduction de ce chapitre, Bird *et al.* [BNM+11] ainsi que Foucault *et al.* [FFB14, FTL+15] mesurent un taux de

1. https://github.com/BroadleafCommerce/BroadleafCommerce

propriété du code au niveau module d'un logiciel. Ici, nous adaptons le travail de Foucault *et al.* [FFB14, FTL+15] sur deux aspects :

- Premièrement, nous proposons une métrique permettant la mesure de la propriété non pas à un niveau module mais à un niveau global. Dans le cas des *contributions architecturales*, cela fait sens car l'architecture concerne l'application dans son ensemble et pas seulement une partie isolée de celle-ci comme pourrait l'être un fichier ou un module.
- Deuxièmement, nous adaptons la métrique permettant la mesure de propriété du code pour l'appliquer au cas spécifique des contributions architecturales. Cette adaptation est basée sur la formalisation définie en Section 5.2.2.

#### 5.2.3.1 Modèle existant de Foucault *et al.*

Foucault *et al.* [FFB14] formalisent la propriété du code pour un développeur $d$ sur un module logiciel $m$ comme suit. Soit:

- $M$ l'ensemble des modules pour une version donnée d'un projet,
- $D$ l'ensemble des développeurs pour cette version du projet,
- $C$ l'ensemble des contributions pour cette version du projet,
- la fonction $files(m)$ donnant l'ensemble des fichiers contenus dans un module $m \in M$.

Il est ensuite possible de définir :

- $w(m_0, d_0)$ le nombre de contributions faites par un développeur $d_0 \in D$ sur un module $m_0 \in M$:

$$w(m_0, d_0) = card(\{c(f, l, n, e, d, t) \in C | d = d_0 \wedge f \in files(m_0)\}) \tag{II.5.2}$$

- $w(m)$ le nombre total de contributions faites sur un module $m \in M$ par l'ensemble des développeurs dans $D$ :

$$w(m) = \sum_{d \in D} w(m, d) \tag{II.5.3}$$

Sur la base de ces éléments, le taux de propriété d'un code pour un développeur $d$ sur module $m$ est défini de la manière suivante :

$$own(m, d) = \frac{w(m, d)}{w(m)} \tag{II.5.4}$$

#### 5.2.3.2 Modèle de mesure du taux global de propriété du code

Précédemment nous avons présenté les éléments du modèle de Foucault *et al.* [FFB14] qui permettent de calculer la propriété du code sur un module logiciel. Nous proposons maintenant une

définition du taux de propriété global du code en s'appuyant sur le formalisme de Foucault *et al.*. Nous avons besoins de redéfinir les fonctions $w$ et $own$ afin qu'elles puissent s'appliquer à l'ensemble du projet pour un développeur $d$ donné. Ainsi nous formalisons :

- $w(d_0)$ comme le nombre de contributions faites par un développeur $d_0$ sur l'ensemble du projet.

$$w(d_0) = card(\{c(f, l, n, e, d, t) \in C | d = d_0\}) \tag{II.5.5}$$

- $w_D$ le nombre de contributions faites par l'ensemble des développeurs dans $D$ sur l'ensemble du projet.

$$w_D = \sum_{d \in D} w(d) \tag{II.5.6}$$

Ensuite, nous définissons $own(d)$ comme le taux de propriété d'un développeur $d$ sur l'ensemble du projet. $own(d)$ est calculé comme un ratio entre l'ensemble des contributions faites par un développeur $d$ sur le projet et l'ensemble des contributions $w_D$ apportées au projet par l'ensemble des développeurs.

$$own(d) = \frac{w(d)}{w_D} \tag{II.5.7}$$

#### 5.2.3.3 Modèle de mesure du taux global de propriété du code architectural

Ayant défini dans la section précédente le taux de propriété du code global $own(d)$ (Équation (II.5.7)), nous allons réutiliser sa définition afin de l'adapter à notre besoin à savoir, la mesure de la propriété du code architectural. Nous réutilisons $own(m)$ mais également la caractérisation du type de contribution expliqué en Section 5.2.2. Nous allons donc définir les éléments suivants :

- $w_a(d_0)$ le nombre de contributions à l'architecture faites par un développeur $d$ sur l'ensemble du projet.

$$w_a(d_0) = card(\{c(f, l, n, e, d, t) \in C | d = d_0 \wedge r(c) = \textsc{Arch}\}) \tag{II.5.8}$$

- $w_{aD}$ le nombre de contributions liées à l'architecture fait par l'ensemble $D$ des développeurs sur l'ensemble du projet.

$$w_{aD} = \sum_{d \in D} w_a(d) \tag{II.5.9}$$

Enfin, comme nous l'avons fait précédemment, nous définissons $own_a(d)$ comme le taux de propriété d'un développeur $d$ sur l'ensemble de l'architecture *runtime* du projet. $own_a(d)$ est calculé comme un ratio entre l'ensemble des contributions à l'architecture *runtime* faites par un développeur $d$ sur le projet et l'ensemble des contributions $w_D$ apportées au projet par l'ensemble des développeurs.

$$own_a(d) = \frac{w_a(d)}{w_{aD}} \tag{II.5.10}$$

Les mesures que nous avons définies ici vont servir dans la suite de ce chapitre pour calculer des taux de propriété et catégoriser les développeurs.

### 5.2.4 Catégorisation des développeurs

Comme nous l'avons indiqué en introduction de ce chapitre, il est possible de catégoriser les développeurs sur la base de leurs taux de propriété du code. Bird *et al.* [BNM+11] ont distingué deux types de développeurs : les *contributeurs majeurs* et les *contributeurs mineurs*. Cette distinction est faite en utilisant un seuil spécifique du taux de propriété du code au-delà duquel les *contributeurs mineurs* deviennent *contributeurs majeurs*. Bird *et al.* ont fixé ce seuil à 5% et ont vérifié sa pertinence à l'aide d'un test de sensibilité. Foucault *et al.* [FFB14, FTL+15] ont réutilisé ce taux dans le cadre de leurs expérimentations sur des projets *open-source*.

#### 5.2.4.1 Catégorisation sur la base du taux global de propriété du code $own(d)$

En réutilisant le seuil de 5% défini par Bird *et al.* [BNM+11] nous catégorisons les développeurs en fonction de leur taux de propriété du code au global ($own(d)$). Tout comme Bird *et al.* nous distinguons deux catégories :

- les *contributeurs mineurs* (abrégés `Cm`) qui se caractérisent par un taux de propriété global inférieur ou égal à 5%. De manière plus formelle, l'ensemble $D_m$ des *contributeurs mineurs* est défini comme suit :

$$D_m = \{d \in D | 0 < own(d) \leq 5\%\} \tag{II.5.11}$$

- les *contributeurs majeurs* (abrégés `CM`) qui se caractérisent par un taux de propriété global strictement supérieur à 5%. Nous définissions l'ensemble $D_M$ des *contributeurs majeurs* comme suit :

$$D_M = \{d \in D | own(d) > 5\%\} \tag{II.5.12}$$

$D_m$ et $D_M$ constituent une partition stricte de la totalité des contributeurs d'un projet : $D = D_m \sqcup D_M$.

#### 5.2.4.2 Catégorisation sur la base du taux global de propriété du code architectural

Pour catégoriser les développeurs et ainsi distinguer ceux s'étant impliqués fortement dans l'architecture des autres, nous réutilisons le seuil de 5% de Bird *et al.* [BNM+11]. Cependant, nous distinguons non plus deux mais trois catégories de développeurs. En effet, la spécificité des contributions architecturales fait qu'il est possible qu'un développeur participe à un projet sans jamais contribuer à son architecture *runtime*. Nous définissons :

- les *contributeurs non-architecturaux* (abrégés `CNAR`) qui n'ont aucune contribution liée à l'architecture. Nous les associons à l'ensemble $A_0$.

$$A_0 = \{d \in D | own_a(d) = 0\} \tag{II.5.13}$$

- les *contributeurs architecturaux mineurs* (abrégés `CARm`) qui ont un taux de propriété global de l'architecture inférieur ou égal à 5%. Nous définissons l'ensemble $A_m$ de ces contributeurs comme suit :

$$A_m = \{d \in D | 0 < own_a(d) \leq 5\%\} \tag{II.5.14}$$

- les *contributeurs architecturaux majeurs* (abrégés `CARM`) qui ont eux un taux de propriété global à l'architecture strictement supérieur à 5%. Nous définissons l'ensemble $A_M$ des *contributeurs architecturaux majeurs* de la manière suivante :

$$A_M = \{d \in D | own_a(d) > 5\%\} \tag{II.5.15}$$

$A_0$, $A_m$ et $A_M$ constituent une partition stricte de l'ensemble des contributeurs : $D = A_0 \sqcup A_m \sqcup A_M$. Ces catégories de contributeurs vont permettre d'explorer des profils de développeurs spécifiques dans un projet logiciel. Elles permettent de donner une vision globale des types de contributeurs dans un projet logiciel.

## 5.3 Questions de recherche

Un projet logiciel est naturellement composé d'une diversité de profils de développeurs. Que l'on se place dans un contexte de projets en sources fermées ou ouvertes il y a une variation du nombre et du type de contribution faites par les développeurs [BNM+11, FFB14, FTL+15]. Les variations s'appliquent aux contributions mais également aux développeurs s'impliquant dans le projet. En effet, un projet *open-source* ou industriel privé est naturellement sujet au *turn-over* de ses participants [RGB06, ICROGB09, FPB+15]. L'implication des développeurs dans le projet est également à mettre en perspective avec l'émergence de nouveaux cadres méthodologiques dits "agiles" [Som15] comme *eXtreme Programming* [Bec99] ou encore *Scrum* [Sch97]. D'après, le principe numéro 11 du Manifeste Agile [FH+01] "*les meilleures architectures, spécifications et conceptions émergent d'équipes auto-organisées*". Au regard des ces éléments, nous pouvons donc nous interroger sur la répartition de la contribution des développeurs sur l'architecture *runtime* des projets logiciels mais également sur le *turn-over*.

**Question de recherche 5.1**

Est-il possible de découvrir des profils de développeurs spécifiques au travers des contributions liées à l'architecture *runtime* d'un logiciel ?

La Question de recherche 5.1 s'intéresse à la possibilité de détecter des profils de développeurs spécifiques en utilisant les métriques et les seuils définis en Section 5.2.

**Question de recherche 5.2**

Existe-t-il un lien entre le taux de propriété au code et celui spécifique à l'architecture *runtime*?

Dans la Question de recherche 5.2 nous questionnons le lien entre les catégories de développeurs *mineurs* et *majeurs* extraites sur la globalité du projet et celles extraites sur l'architecture *runtime*.

**Question de recherche 5.3**

Quel est le *turn-over* entre les différentes catégories de développeurs ?

Notre troisième question de recherche (Question de recherche 5.3) s'intéresse au *turn-over* potentiellement observable entre des catégories de développeurs. Cette question interroge les mouvements entrants et sortants de l'ensemble des développeurs du projet mais également les changements de catégorie des développeurs.

## 5.4 Approche proposée

Nous avons modélisé en Section 5.2 la contribution d'un développeur, les types de contributions mais également défini les métriques permettant l'analyse des taux de propriété du code.
Dans cette section, nous détaillons le contexte expérimental mais également le processus d'analyse. Nous présentons également le cas d'étude sur lequel nous conduisons nos expérimentations.

### 5.4.1 Cas d'étude : *BroadleafCommerce*

Notre méthodologie est appliquée sur un projet industriel *open-source* et disponible sur GitHub: *BroadleafCommerce*[2]. Ce projet est une application Java dont l'architecture *runtime* est conçue à l'aide du *framework Spring*[3]. *BroadleafCommerce* est un *framework* dédié à la création de sites marchands en ligne. C'est un projet populaire sur GitHub avec plus de 1500 "*stars*", un millier de

2. https://github.com/BroadleafCommerce/BroadleafCommerce
3. https://spring.io

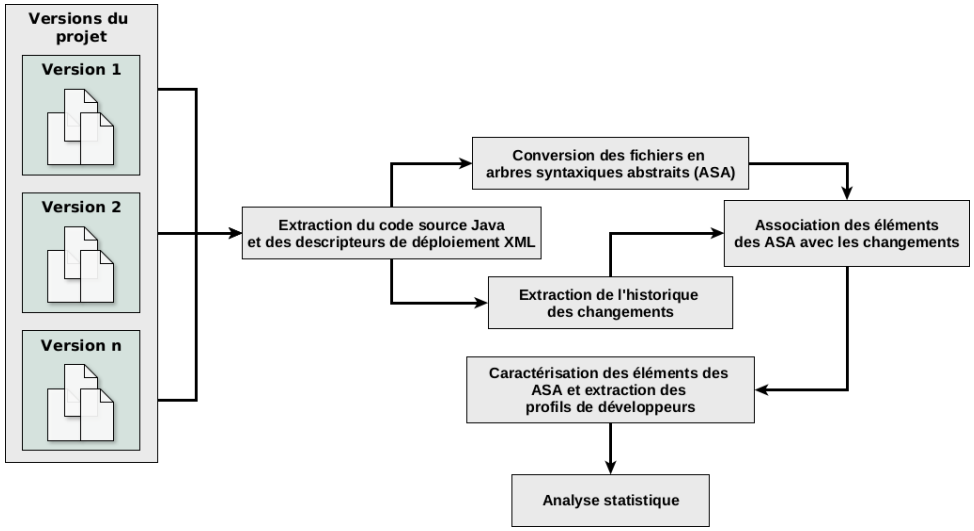



FIGURE II.5.1 – Processus d'extraction et d'analyse des contributions des développeurs.

*forks* et plus de 70 développeurs. De fait, ce projet satisfait les critères de qualité pour conduire une étude selon Kalliamvakou *et al.* [KGB+16]. Dans le cadre de nos travaux, nous analysons 13 versions étiquetées *General-Availability* (*GA*). Ces 13 versions correspondent à des versions de production et couvrent sept ans de vie du projet.

### 5.4.2 Contexte expérimental et processus d'analyse

L'approche proposée pour l'extraction de données et l'analyse est présentée dans la Figure II.5.1. Les fichiers du projet sont premièrement extraits du dépôt logiciel. L'ensemble du code source et des fichiers correspondants à des descripteurs de déploiement de l'architecture *runtime* sont analysés pour chaque version du projet. Deux traitements indépendants sont ensuite effectués en parallèle :

- la recherche de l'historique des changements par les développeurs sur chaque lignes des fichiers est recherché
- l'analyse syntaxique de l'ensemble des fichiers afin de transformer l'ensemble du contenu du fichier en arbre syntaxique abstrait (ASA).

L'étape suivante consiste à associer chaque élément de l'arbre syntaxique abstrait aux changements effectués sur chaque ligne afin d'identifier les contributions qui se rapportent à la définition de l'architecture *runtime* (analyse sémantique). Les métriques $own(d)$ (Équation (II.5.7)) et $own_a(d)$ (Équation (II.5.10)) sont ensuite calculées pour chaque développeur. Enfin, une analyse statistique recherche les différentes catégories des développeurs.
Nous avons implémenté ce processus en utilisant les technologies suivantes :

- *Git* comme système de gestion de versions

- *Git-Blame* pour rechercher l'historique de modification des fichiers
- *Java Development Tool* (*JDT*) pour analyser grammaticalement les fichiers de code ainsi que les descripteurs de déploiement
- *Python* et *R* pour extraire les profils des développeurs et faire les analyses statistiques.

Par ailleurs, les mesures liées au taux de propriété architecturale sont basées sur les fonctionnalités de déploiement de l'architecture *runtime* fournies par le *framework Spring*. Notons qu'aucun de ces choix de technologies n'est restrictif : ceux-ci sont basés sur des principes généraux et peuvent donc être adaptés à d'autres écosystèmes (pour ce qui concerne par exemple *Git* et *Git-Blame*) ou avoir un équivalent (pour ce qui concerne par exemple *framework Spring* ou *JDT*).
L'extraction des différents profils de développeurs se fait en utilisant le processus décrit dans la section précédente. Ce processus est découpé en trois étapes majeures :

1. Extraction des contributions (comme modélisé en Section 5.2.1) en utilisant *Git-Blame*.
2. Catégorisation des contributions en fonction de leur type : contribution participant à l'architecture *runtime* ou non (modélisé en Section 5.2.2). Dans notre cas d'étude, deux types de fichiers contiennent de l'information à propos de l'architecture *runtime* Spring :
   - les fichiers Java contenant des annotations permettant de déclarer ou d'utiliser des composants. Les annotations : `@Component`, `@Repository`, `@Controller`, `@RestController`, `@Configuration`, `@Bean`, `@Service`, `@Autowired` sont utilisées et combinées pour créer des architectures *runtime* avec le *framework Spring*. Ces annotations sont considérés comme les marqueurs sémantiques permettant d'identifier les contributions à l'architecture *runtime* dans les fichiers de code Java.
   - les descripteurs de déploiement XML Spring qui utilisent des balises spécifiques pour définir l'architecture *runtime* selon le *framework Spring*. Les balisent utilisées pour créer une architecture sont `<beans>` et `<bean>`. Elles appartiennent également à l'ensemble des marqueurs architecturaux.

   Ainsi dans notre étude l'ensemble $M_a$ des marqueurs architecturaux est le suivant :

   $M_a = \{$`@Component`, `@Repository`, `@Controller`, `@RestController`, `@Configuration`, `@Bean`, `@Service`, `@Autowired`, `<beans>`, `<bean>`$\}$

3. Pour chaque version du projet, les taux de propriété du code $own(d)$ et $own_a(d)$ sont calculés pour chaque développeur ayant contribué au projet (tel que modélisé dans les Sections 5.2.3.2 et 5.2.4.1).

## 5.5 Résultats

Dans cette section, nous présentons puis analysons qualitativement les résultats liés aux différentes questions présentées en Section 5.3. En premier lieu, nous étudions les différents profils de développeurs qu'il est possible d'extraire à l'aide de nos métriques. Puis, nous étudions le lien entre les

taux de propriété globale et architecturale. Enfin, nous étudions le *turn-over* et les changements de catégorie des développeurs.

### 5.5.1 Profils de développeurs et contributions architecturales

Nous avons émis dans la Question de recherche 5.1 l'idée que des types différents de profils de développeurs pouvaient être mis en lumière en se basant sur une métrique d'évaluation du taux de propriété du code dans sa globalité $own(d)$ (Équation (II.5.7)) et du taux global de propriété de l'architecture *runtime* $own_a(d)$ (Équation (II.5.10)). Nous avons utilisé la métrique $own(d)$ pour la catégorisation des développeurs en *contributeurs mineurs* et *contributeurs majeurs* selon le seuil de 5% (Équations (II.5.11) et (II.5.12)). De manière analogue, en utilisant $own_a(d)$ nous avons catégorisé les développeurs contribuant à l'architecture *runtime* en trois catégories:

- les *contributeurs non-architecturaux* qui n'ont pas de contributions à l'architecture *runtime* (Équation (II.5.13)),
- les *contributeurs architecturaux mineurs* qui ont un taux global de propriété de l'architecture *runtime* inférieur ou égale à 5% (Équation (II.5.14)),
- enfin les *contributeurs architecturaux majeurs* ayant un taux global de propriété de l'architecture de plus de 5% (Équation (II.5.15)).

La Figure II.5.2 montre les pourcentages des contributions à l'architecture *runtime* pour chaque développeur et chaque version du projet *BroadleafCommerce*. Sur la Figure II.5.2 un point correspond à un développeur et chaque courbe à une version étudiée du projet. Afin de mieux visualiser les tendances se dégageant du graphe, les développeurs sont triés par ordre décroissant de leur taux de propriété architecturale pour l'ensemble des versions. Nous constatons que pour l'ensemble des versions la métrique $own_a(d)$ sépare les développeurs en trois groupes :

- les développeurs ayant un taux de propriété du code architectural égal à 0 (à des fins de lisibilité, ceux-ci ne sont pas représentés sur la Figure II.5.2).
- les développeurs ayant un très faible taux de propriété du code architectural et donc peu de contributions à l'architecture. Ces développeurs sont visibles sur la partie plate des courbes.
- les développeurs ayant un fort taux de propriété du code architectural se traduisant ici par un apport important de contributions à l'architecture *runtime* du projet. Ces développeurs sont représentés par la partie gauche montante sur la figure.

Cette visualisation donnée par la Figure II.5.2 permet de distinguer clairement des catégories de développeurs, ainsi, elle conforte le notamment le seuil de 5% Bird *et al.* [BNM+11] et la catégorisation que ce dernier permet.

Les Figures II.5.3 et II.5.4 présentent les résultats sur les différentes catégories de développeurs que nous avons analysés à savoir, les *contributeurs architecturaux majeurs* et *contributeurs majeurs* (`CAMR_CM`), les *contributeurs architecturaux majeurs* et *contributeurs mineurs* (`CAMR_Cm`), les *contributeurs architecturaux mineurs* et *contributeurs majeurs* (`CAmR_CM`), les *contributeurs architecturaux mineurs* et *contributeurs mineurs* (`CAmR_Cm`) et les *contributeurs non-architecturaux* et

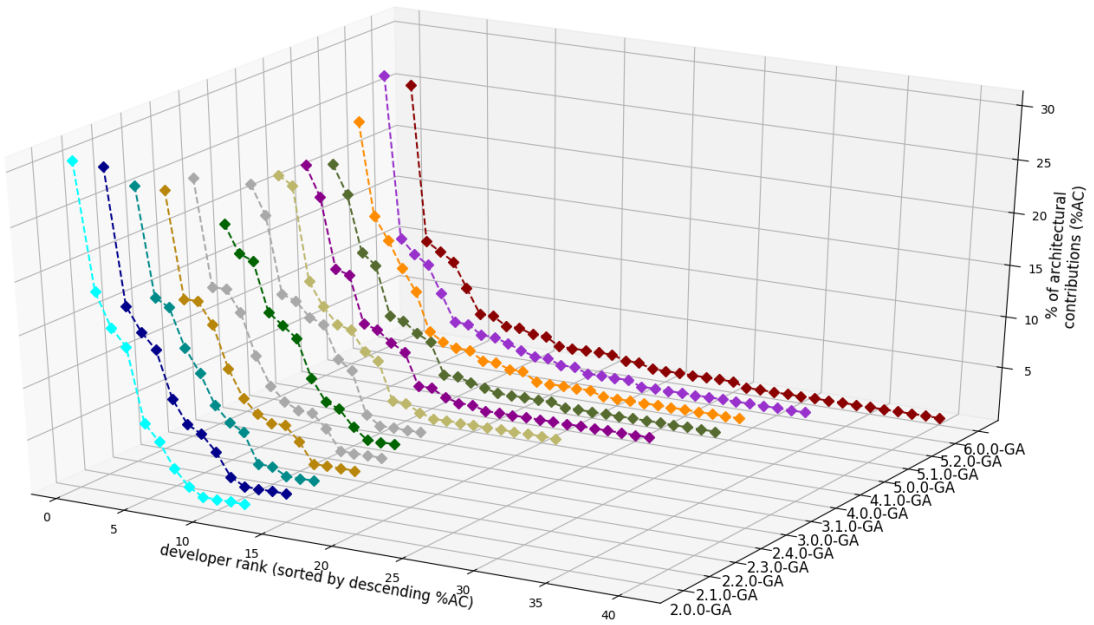



FIGURE II.5.2 – Pourcentages de contributions à l'architecture *runtime* par version du projet *BroadleafCommerce* et par développeurs.

*contributeurs mineurs* (`CNAR_Cm`). Nous constatons que les combinaisons de catégories construites à l'aide des métriques $own_a$ et *own* permettent de dissocier différents profils de développeurs. Nous observons cependant l'absence de développeurs dans la catégorie *contributeurs non-architecturaux* et *contributeurs majeurs* (`CNAR_CM`). Cette absence peut s'expliquer par le fait que les développeurs dits *contributeurs architecturaux majeurs* de par leur expérience et leurs contributions ont forcément été amenés à manipuler du code lié à l'architecture *runtime*. Les *contributeurs majeurs* sont également des développeurs ayant une vision globale du projet. Ils sont donc à même d'effectuer des modifications sur l'architecture *runtime*. Nous constatons que la catégorie des *contributeurs architecturaux majeurs* et *contributeurs majeurs* (`CAMR_CM`) est relativement stable tout au long des versions. Un pool de développeurs expérimentés ayant la connaissance de l'ensemble de l'application contribue certainement au cœur du projet en le maintenant et l'améliorant durant sa vie. Cette observation peut être interprétée comme un indice d'une bonne gestion du projet par *BroadleafCommerce*. Un grand nombre de développeurs sont et restent des *contributeurs architecturaux mineurs* ou *contributeurs non-architecturaux* (`CNAR_Cm`).

Ce constat peut provenir de la nature ouverte du projet analysé. En effet, dans les projets *open-source* certains développeurs participent de manière très ponctuelle voir de manière unique. Il n'est donc pas anormal de faire ce type d'observation ici.

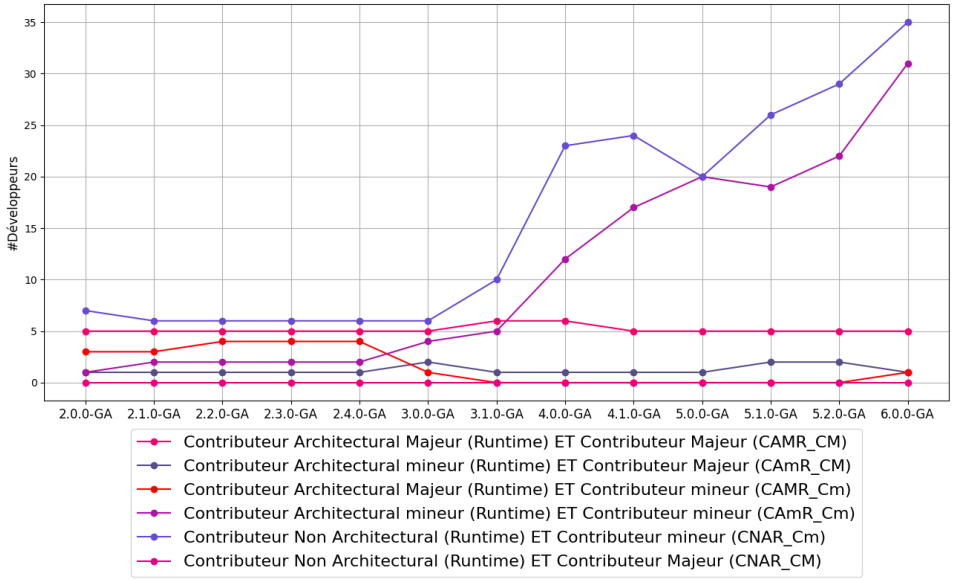



FIGURE II.5.3 – Nombre de développeurs par catégorie pour chaque version du projet *Broadleaf-Commerce*.

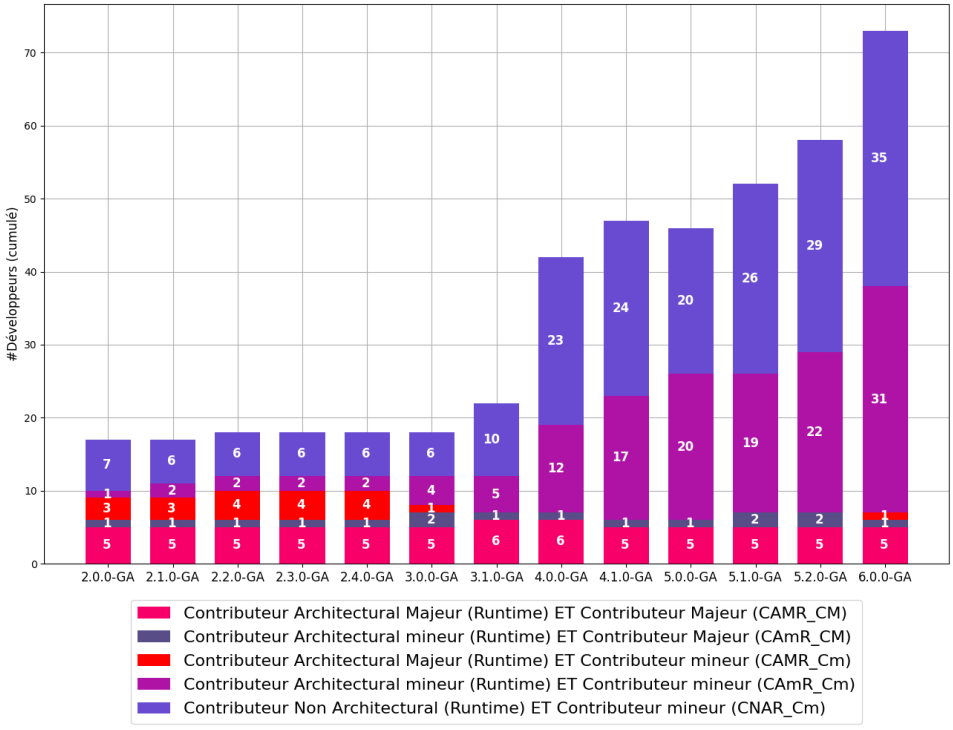



FIGURE II.5.4 – Nombre de développeurs cumulés par catégorie pour chaque version du projet *BroadleafCommerce*.

### 5.5.2 Lien entre taux de propriété globale et taux de propriété architecturale

Nous avons montré précédemment la faisabilité de la catégorisation des développeurs au travers de métriques spécifiques à l'architecture. La Question de recherche 5.2 s'intéresse au lien potentiel entre le taux de propriété global mesuré par $own(d)$ et celui spécifique à l'architecture. Notre métrique $own_a(d)$ étant basée sur les contributions spécifiques à l'architecture, il est intéressant d'étudier la relation qu'elle peut avoir avec le taux de propriété globale mesuré par $own(d)$. Les Figures II.5.3 et II.5.4 montrent que les *contributeurs non-architecturaux* sont exclusivement des *contributeurs mineurs*. Cependant, l'implication opposée n'est pas vraie. En effet, les *contributeurs mineurs* sont répartis de manière relativement égale entre les développeurs ne participant pas à l'architecture *runtime* et ceux s'y impliquant. Les développeurs *contributeurs mineurs* (`Cm`) qui sont aussi des *contributeurs architecturaux majeurs* (`CAMR`) existent également et sont majoritairement présents jusqu'à la version 3.0.0-GA de *BroadleafCommerce*. Dans le même temps, les *contributeurs majeurs* (`CM`) sont généralement des *contributeurs architecturaux majeurs*.
Pour analyser la situation de manière objective nous avons mené un test statistique basé sur les hypothèses suivantes :

- $\mathcal{H}_0$*: Être un contributeur architectural majeur (`CAMR`) est indépendant du fait d'être un contributeur majeur (`CM`) ?* (hypothèse nulle)
- $\mathcal{H}_1$*: Être un contributeur architectural majeur (`CAMR`) est dépendant du fait d'être un contributeur majeur (`CM`) ?* (hypothèse alternative)

Afin de valider ou d'invalider ces hypothèses nous avons utilisé un test de Fisher au risque $\alpha = 0.05$. Le test est significatif quand la valeur $p < \alpha$. Le test de Fisher est choisi car il permet de mesurer le lien statistiques entre 2 variables qualitatives sur des échantillons de population de faibles tailles, ce qui est notre cas.

| version | 2.0.0-GA | 2.1.0-GA | 2.2.0-GA | 2.3.0-GA | 2.4.0-GA | 3.0.0-GA | 3.1.0-GA |
|---|---|---|---|---|---|---|---|
| *p-value* | 1 | 0.54545 | 1 | 1 | 1 | 0.24242 | **0.01515** |

| version | 4.0.0-GA | 4.1.0-GA | 5.0.0-GA | 5.1.0-GA | 5.2.0-GA | 6.0.0-GA |
|---|---|---|---|---|---|---|
| *p-value* | $\mathbf{4.5566^{-6}}$ | **0.00017** | $\mathbf{9.12131^{-5}}$ | **0.00031** | **0.00017** | $\mathbf{6.99102^{-5}}$ |

TABLE II.5.1 – Résultats du test de Fisher sur les 13 versions du projet *BroadleafCommerce*

La Table II.5.1 présente les résultats du test de Fisher sur les 13 versions analysée du projet *BroadleafCommerce*. Les valeurs de $p$ significatives sont en **gras** dans la Table II.5.1. Les tests confirment qu'à partir de la version 3.1.0-GA il existe une relation statistique entre le fait d'être un *contributeur architectural majeur* (`CAMR`) et celui d'être un *contributeur majeur* (`CM`) et nous pouvons donc confirmer l'hypothèse $\mathcal{H}_0$. Pour les versions antérieurs le lien statistique n'est pas établi. Cette non-significativité est principalement attribuée à la présence des développeurs *contributeurs architecturaux majeurs* et *contributeurs mineurs* `CAMR_mC` avant la version 3.1.0-GA.

### 5.5.3 *Turn-over* et catégories de développeurs

Après avoir montré en que la métrique $own_a(d)$ est pertinente et qu'elle permet de déterminer des catégories de développeurs, nous allons regarder le *turn-over* qui existe entre de celles-ci. Les Figures II.5.3 et II.5.4 ont montré une évolution intéressante : l'effectif de développeurs *contributeurs architecturaux majeurs* et *contributeurs majeurs* (`CARM_CM`) reste stable dans le temps tandis que celui des *contributeurs architecturaux majeurs* et *contributeurs mineurs* (`CARM_Cm`) devient nul. La catégorie des *contributeurs non-architecturaux* augmente de manière importante après la version 3.1.0-GA. Ainsi, pour mieux comprendre les mouvements entre catégories, nous analysons le *turn-over* entre deux versions successive pour chaque version de *BroadleafCommerce*. Nous ajoutons une catégorie *Externe* correspondant aux développeurs entrants et sortants du projet.

La Figure II.5.5 montre les résultats obtenus sous la forme de diagrammes de cordes. Une catégorie est représentée par un arc sur le bord du cercle. Un changement de catégorie est, quant à lui, matérialisé par une flèche partant de la catégorie source vers la catégorie cible. La proportion des changements est définie par la largeur du lien entre deux catégories. La population d'une catégorie inchangée est représentée par une région intérieure colorée. Par exemple, entre les versions 2-0-0-GA et 2-1-0-GA, une partie des développeurs *contributeurs non-architecturaux* et *contributeurs mineurs* (`CNAR_Cm`) a migré vers la catégorie *contributeurs architecturaux mineurs* et *contributeurs mineurs* (`CARm_Cm`). Comme montré par les Figures II.5.3 et II.5.4, la catégorie des développeurs dits *contributeurs non-architecturaux* et *contributeurs architecturaux majeurs* étant vide pour l'ensemble des versions, elle n'est pas représentée sur la Figure II.5.5.
La Figure II.5.5 montre un très faible *turn-over* pour les catégories de développeurs *contributeurs architecturaux majeurs* et *contributeurs majeurs* (`CARM_CM`). Cela révèle une stabilité de la population de développeurs dans cette catégorie. Les quelques nouveaux membres de cette catégorie proviennent de la catégorie *contributeurs architecturaux majeurs* et *contributeurs mineurs* (`CARM_Cm`) mais aussi de celle des *contributeurs architecturaux mineurs* et *contributeurs majeurs* (`CARm_CM`). La provenance de ces développeurs indique que seuls des développeurs expérimentés semblent entrer dans la catégorie des `CARM_CM`. Cela pourrait à nouveau confirmer l'existence d'une politique de gestion du projet par un noyau de développeurs centraux ayant suffisamment de compétences et de connaissances pour modifier l'architecture *runtime*. Cela est d'autant plus vrai qu'en tant que projet industriel, *BroadleafCommerce* est probablement dirigé par une petite équipe professionnelle correspondant à la catégorie `CARM_CM` détectée ici. La grande partie des développeurs `CARM_Cm` migrent vers la catégorie des *contributeurs architecturaux mineurs* et *contributeurs mineurs* (`CARm_Cm`) dans la version 3.0.0-GA. Le reste des membres de cette catégorie devient membre de la catégorie des `CARM_CM` en version 3.1.0-GA. Cela pourrait indiquer que rester `CARM_CM` sur le long terme implique d'être ou d'avoir appartenu à la catégorie des *contributeurs majeurs*. Cependant, la stabilité et le faible *turn-over* de la catégorie `CARm_Cm` dans les premières versions de même que sa réapparition dans la version 6.0.0-GA pourrait suggérer un besoin ponctuel et spécifique d'un développeur expérimenté sur l'architecture *runtime*. Un autre constat est que l'ensemble

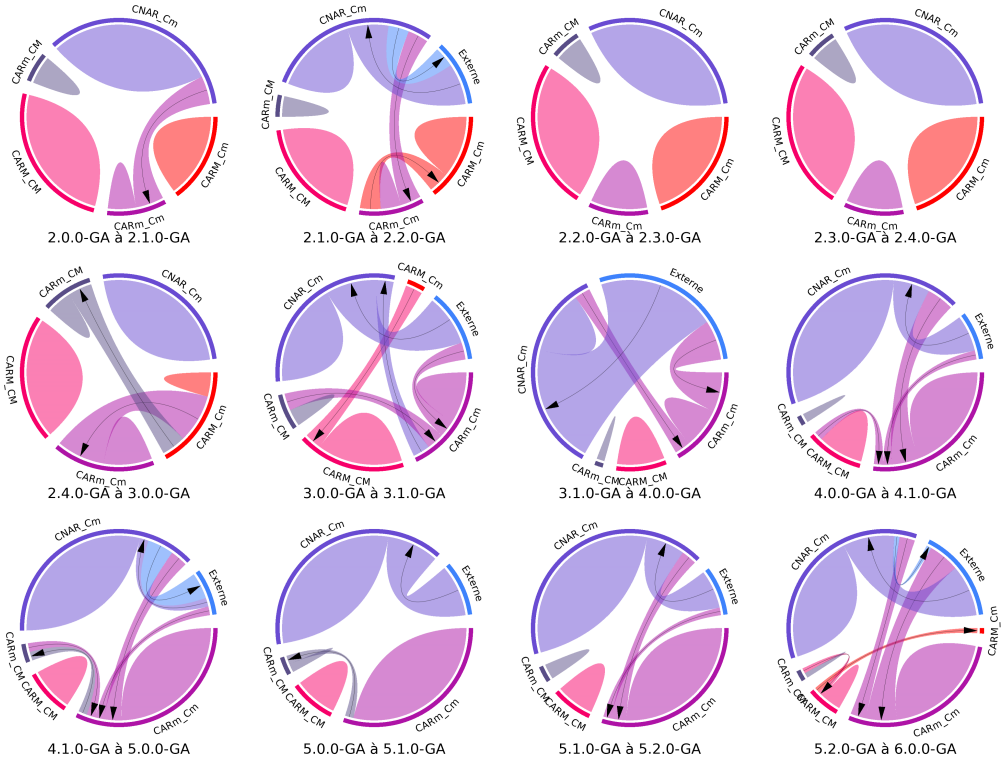



FIGURE II.5.5 – Diagrammes cordes illustrant les changements de catégories de contributeurs entre 2 versions du projet *BroadleafCommerce*.

des développeurs provenant de l'extérieur rejoignent principalement la catégorie des *contributeurs non-architecturaux* et jamais celle des *contributeurs architecturaux majeurs*. Enfin, beaucoup de *contributeurs non-architecturaux* ne deviennent jamais des *contributeurs architecturaux mineurs*.

## 5.6 Synthèse, limites et perspectives

Dans ce chapitre nous avons créé un modèle formel de mesure du taux de propriété globale du code mais aussi de l'architecture *runtime*. En premier lieu, nous avons montré sur notre cas d'étude la pertinence de nos métriques. Puis, nous avons extrait des catégories de développeurs à l'aide de celles-ci. Enfin, nous avons mesuré le *turn-over* des développeurs entre les catégories.

### 5.6.1 Synthèse

**Synthèse Question de recherche 5.1**

Nos observations de la Section 5.5.1 montrent que la métrique $own_a(d)$ (Équation (II.5.10)) que nous proposons est pertinente pour mesurer le taux de propriété du code source lié à l'architecture *runtime*. Elle démontre son efficacité pour définir des profils de développeurs. Malgré son caractère simple, elle permet de détecter de manière consistante trois catégories de contributions au code source de l'architecture *runtime*: les *contributeurs majeurs*, les *contributeurs mineurs* et les *non-contributeurs*. Il est à noter que nous avons été capables de catégoriser les développeurs sur l'ensemble des versions, ce qui équivaut à 7 ans de vie d'un projet dont la taille a été croissante. La simplicité de notre métrique est probablement contrebalancée par la nature spécifique de la mesure à savoir le taux de propriété de l'architecture *runtime*. Il en résulte une métrique suffisamment sensible pour détecter les différentes catégories de développeurs recherchées et suffisamment robuste pour isoler un ensemble de développeurs architecturaux majeurs quasiment stable dans le temps, ce qui tendrait à monter une gestion active du projet.

**Synthèse Question de recherche 5.2**

Ces résultats de la Section 5.5.2 permettent de donner une réponse positive à la Question de recherche 5.2. Comme attendu, les *contributeurs majeurs* sont généralement des *contributeurs architecturaux majeurs*. Cette conclusion fait sens puisque les développeurs les plus expérimentés sont en général ceux qui contribuent beaucoup au projet et sont donc en capacité de s'occuper de parties sensibles comme l'architecture *runtime*. Il est intéressant d'observer que les *contributeurs architecturaux majeurs* peuvent être aussi des *contributeurs mineurs*. Cela vient conforter la pertinence de notre métrique car être *contributeur architectural majeur* n'est pas qu'une simple conséquence au fait d'être un *contributeur majeur*. Au contraire, être *contributeur architectural* sur l'architecture *runtime* est un rôle délibérément choisi. En effet, les *contributeurs non-architecturaux* sont présents systématiquement dans l'ensemble des versions. De plus, être un *contributeur majeur* architectural ou non semble correspondre à un rôle encore plus spécifique, choisi et soutenu activement et non pas à un rôle acquis passivement puisque l'effectif de la catégorie est globalement stable sur l'historique du projet.

**Synthèse Question de recherche 5.3**

Les observations faites en Section 5.5.3 permettent de répondre à la Question de recherche 5.3 par la confirmation de la pertinence de nos métriques et des catégories observables. En effet, nous avons été capables de séparer les différentes catégories de développeurs et d'observer les changements à travers le temps. Nous avons observé qu'il existait un noyau de développeurs expérimentés contribuant fortement au projet et à l'architecture *runtime*. De plus, il semble que faire partie de ce groupe de développeurs est un choix délibéré ou le résultat d'une gestion effective du rôle des contributeurs au sein du projet. Une hiérarchisation des catégories de développeurs semble se dégager : les développeurs entrants ne sont pas immédiatement des *contributeurs architecturaux majeurs*. Ils intègrent d'abord des catégories intermédiaires.

### 5.6.2 Incertitudes sur la validité

#### Validité de la construction

Les architectures logicielles sont gérées à différents niveaux : modules, code source ou comme ici *runtime*. Elles sont dépendantes des technologies ou des choix conceptuels faits par les membres du projet. Les contributions architecturales sont donc variables et multi-facettes. La métrique $own_a(d)$ proposée dans cette section détecte les développeurs contribuant de manière importante à l'architecture *runtime*. Nous ne la considérerons pas comme une manière unique de profiler un développeur, mais plutôt comme un point d'entrée au profilage d'architectes et d'analyse des politiques de gestion de l'architecture.

#### Validité externe

Concernant la validité externe, nos résultats se limitent à une preuve de concept effectuée sur un projet unique. Pour extraire une connaissance empirique, il est nécessaire d'analyser un grand nombre de projets. De plus, nous analysons des projets utilisant le *framework* Spring mais il serait nécessaire de généraliser à d'autres *frameworks* comme Apache Struts, OSGI ou encore JEE. Cependant, afin de limiter les incertitudes quant à la sélection du projet parmi ceux disponibles sur GitHub, nous avons respecté les critères donnés par Kalliamvakou *et al.* [KGB+16]. La répartition des développeurs dans *BroadleafCommerce* pourrait être une singularité mais il y a de grandes chances que des résultats comparables puissent être extraits d'autres projets ayant un niveau de qualité comparable. Un biais pourrait être introduit dans nos résultats par le langage de programmation utilisé (ici Java). Un second biais pourrait être lié à l'utilisation d'un *framework* spécifique qui peut induire aussi des paradigmes (services web) ou des applications spécifiques (sites web). Une étude avec un panel de projets dans d'autres langages comme Javascript ou Go pourrait être envisagée afin d'éliminer ce biais.

#### Validité interne

Dans un soucis d'économie du temps de calcul, nous avons volontairement échantillonné les versions de *BroadleafCommerce* en choisissant celles ayant un tag *General Availability*. Cela pourrait constituer un biais, bien que nos mesures aient été faites sur l'historique complet des changements.

#### Fiabilité

Les résultats présentés ici ont été vérifiés par deux implémentations différentes du processus de récupération de données. Cependant, la présence de bogues résiduelles ne peut être formellement exclue.

#### Incertitudes statistiques

L'incertitude statistique concerne la confiance et la pertinence accordées aux résultats. Dans notre test de Fisher, nous utilisons un risque $\alpha$ de 5%. En considérant ce niveau de risque, nous avons 5% de chance de trouver un résultat identique avec des valeurs aléatoires. Cependant, ce niveau de risque est communément admis dans d'autres études empiriques.

### 5.6.3 Limites

Un point d'attention dans nos travaux vient du seuil de Bird *et al.* [BNM+11] que nous réutilisons pour catégoriser les contributeurs par rapport à leur contribution globale (*own*). En effet, ce seuil a été défini pour des contributions mesurées sur des éléments de faible granularité (librairies). Or nous l'utilisons de manière globale sur l'ensemble du code dans un projet qui grandit au fil du temps. Ainsi, les contributions des développeurs sont réparties dans l'ensemble du projet ce qui implique qu'un contributeur, pour être majeur, doit mécaniquement contribuer à une grande partie du code. De cette manière, il se peut que le seuil de 5% soit plus sévère appliqué globalement sur le projet que sur une unité du projet. Cela aurait pour effet de réduire de le nombre de *contributeurs majeurs* et augmenter de le nombre de *contributeurs mineurs*

### 5.6.4 Perspectives

Une première perspective serait de conduire une étude similaire sur un large éventail de projets de même nature que celui expertisé ici, à savoir, des projets utilisant le langage Java et le *framework* Spring. Un élargissement à d'autres langages de programmation et à d'autres *frameworks* d'architectures est également envisageable. Cela permettrait de conforter les résultats obtenus ici mais aussi d'étudier les politiques de management des projets en fonction de leurs technologies.
Une deuxième perspective pourrait être la mesure du taux de propriété et des catégories de développeurs associés sur d'autres aspects architecturaux. Cela permettrait de comparer les similarités et les disparités entre catégories de développeurs contribuant à différentes formes d'architectures.

De ce fait, il serait envisageable d'étudier par exemple les développeurs contribuant au code Java et ceux contribuant à l'architecture dite architecture *runtime* construite à l'aide du *framework* Spring.
Enfin, une troisième perspective porterait sur l'étude de "points chauds" dans le projet. Le taux de propriété architecturale et les catégories pourraient être mesurés puis associés aux différents éléments du projet. Ainsi, il serait possible de visualiser si des éléments du projets subissent ou non une forte contribution architecturale et par quel type de développeurs. De plus, il serait potentiellement possible de quantifier le nombre nécessaire de développeurs expérimentés nécessaire pour l'architecture par rapport à la taille d'un ou plusieurs éléments architecturaux ce qui pourrait constituer une règle de bonne pratique dans la conduite de projet.

# Chapitre II.6

# Classification automatique de développeurs expérimentés dans des projets *open-source*

**Sommaire**

## 6.1 Introduction

L'expérience du développeur est à la fois subjective et relative. Subjective car elle dépend de la manière dont se considère le développeur dans son métier, de la façon dont il a construit ses connaissances et compétences mais aussi de son environnement de travail. Relative car, au-delà de l'expérience purement technique acquise au fil du temps, le développeur doit d'acquérir, pour chaque projet, une expérience spécifique. Recenser les personnes expérimentées dans un projet logiciel n'est pas une tâche aisée. A ce sujet, Izquierdo *et al.* [IROG09] montrent que le *turn-over* dans les projets *open-source* peut conduire à des problèmes de management de projet, d'autant plus critiques quand les développeurs expérimentés sont concernés. Dans le Chapitre II.5, nous avons montré qu'il est possible de repérer des catégories de développeurs spécifiques, notamment des développeurs contribuant à l'architecture *runtime*, par dénombrement d'éléments spécifiques dans le code source. Cependant, selon Kruchten [Kru99], les architectes des projets sont souvent des développeurs expérimentés capables de manipuler un vaste panel de technologies et de concepts. La mesure que nous avons proposée dans le Chapitre II.5 est trop spécifique pour détecter l'ensemble des architectes contribuant au développement. Une approche plus globale permettant de détecter les développeurs expérimentés, et donc les architectes potentiels, est donc à considérer.

L'objectif de ce chapitre est de montrer la faisabilité et la performance d'une approche permettant de détecter les développeurs expérimentés en utilisant de l'apprentissage supervisé. En Section 6.3, nous présentons globalement l'approche que nous avons mise en place. En Section 6.4, nous détaillons notre approche comprenant la création du jeu de données, les traitements effectués sur ces données puis la comparaison de différents classifieurs supervisés. En Section 6.5, nous présentons et commentons les résultats de notre approche. Enfin, en Section 6.6 nous présentons une synthèse de nos travaux ainsi que leurs limites et perspectives.

## 6.2 Questions de recherche

Nous avons vu en introduction que le *turn-over* des développeurs expérimentés peut entraîner des problèmes de gestion du projet [IROG09]. De plus, l'identification de ces développeurs n'est pas une tâche aisée. Dans le Chapitre I.3, nous avons vu que plusieurs types d'approches coexistent : les approches par regroupement et les approches par profilage. Les approches par regroupement cherchent à grouper les développeurs en catégories à partir de leurs caractéristiques. Les approches par profilage permettant d'identifier un ou plusieurs développeurs en fonction de leurs caractéristiques, c'est sur ce type d'approche que nous allons nous focaliser. Aucune de ces approches n'utilise de l'apprentissage supervisé. Ainsi, à partir de ces éléments, nous formulons deux questions de recherche.

**Question de recherche 6.1**

Peut-on créer un classifieur supervisé suffisamment performant pour profiler les développeurs expérimentés sur la base de métriques logicielles ?

La Question de recherche 6.1 interroge la possibilité de disposer d'un classifieur utilisant un algorithme d'apprentissage supervisé et permettant le profilage de développeurs expérimentés, en utilisant directement des métriques logicielles. Ce type de méthode, qui combine à la fois apprentissage supervisé et métriques, n'étant pas répertorié dans l'état de l'art, il nous semble intéressant de l'étudier.

**Question de recherche 6.2**

Quelles sont les variables des développeurs qui orientent le choix du classifieur ?

La Question de recherche 6.2 interroge l'explicabilité du classifieur et sa manière d'orienter la classification en fonction de l'impact des variables utilisées pour cette dernière. Pour répondre à cette question, nous utiliserons une méthode d'explicabilité (*Explainable AI*) permettant de calculer et de visualiser l'influence des variables sur la classification.

## 6.3 Approche globale

Dans cette section, nous présentons une vision globale de l'approche proposée puis nous détaillerons chaque étape dans la section suivante. Notre approche est découpée en trois étapes principales : la création du jeu de données, le traitement des variables et la sélection d'un classifieur.

### Création du jeu de données

Les développeurs sont extraits de 17 projets *open-source* qui sont hébergés sur GitHub. La Table II.6.1 donne la liste des 17 projets utilisés qui ont été choisis selon les critères de qualité définis par Kalliamvakou *et al.* [KGB+16]. L'ensemble des développeurs participant à ces projets est extrait puis des métriques logicielles sont calculées. Pour chacun d'entre eux, nous étiquetons ensuite le jeu de données en recherchant sur internet des informations à propos du rôle que tient chaque développeur dans un projet. Enfin, nous nettoyons le jeu de données par vérification manuelle des valeurs aberrantes et nous l'anonymisons.

| Projet | #Contributeurs | #Stars | Date de création | URL |
|---|---|---|---|---|
| Activiti | 152 | 8012 | 2012-09 | https://github.com/Activiti/Activiti |
| BroadleafCommerce | 61 | 1490 | 2011-12 | https://github.com/BroadleafCommerce/BroadleafCommerce |
| Camunda-bpm-platform | 29 | 2203 | 2013-01 | https://github.com/camunda/camunda-bpm-platform |
| Dhis2-core | 64 | 210 | 2016-08 | https://github.com/dhis2/dhis2-core |
| Flowable-engine | 199 | 4385 | 2016-10 | https://github.com/flowable/flowable-engine |
| Jetcache | 11 | 3035 | 2017-04 | https://github.com/alibaba/jetcache |
| Moduliths | 7 | 568 | 2018-05 | https://github.com/odrotbohm/moduliths |
| Piggymetrics | 13 | 10820 | 2015-03 | https://github.com/sqshq/piggymetrics |
| Problem-spring-web | 17 | 734 | 2015-08 | https://github.com/zalando/problem-spring-web |
| Spring-boot-admin | 94 | 10183 | 2014-07 | https://github.com/codecentric/spring-boot-admin |
| Spring-petclinic | 47 | 7454 | 2013-01 | https://github.com/spring-projects/spring-petclinic |
| Spring-social | 27 | 618 | 2011-02 | https://github.com/spring-projects/spring-social |
| Spring-social-facebook | 23 | 242 | 2011-05 | https://github.com/spring-projects/spring-social-facebook |
| Spring-social-linkedin | 7 | 71 | 2011-05 | https://github.com/spring-projects/spring-social-linkedin |
| Springfox | 128 | 5330 | 2012-05 | https://github.com/springfox/springfox |
| UPortal | 66 | 222 | 2011-10 | https://github.com/uPortal-Project/uPortal |
| Ureport | 6 | 1432 | 2017-06 | https://github.com/youseries/ureport |

TABLE II.6.1 – Liste des 17 projets utilisés pour constituer le jeu de données de développeurs.

#### Traitement des variables

Les données récupérées n'étant pas standardisées et étant probablement bruitées, une étape de traitement et de nettoyage (*feature engineering*) est nécessaire. Cette étape va permettre la suppression des données engendrant du bruit, la mise à l'échelle des données et la réduction de la dispersion des valeurs.

#### Sélection d'un classifieur

Différents types de classifieurs existent et nous souhaitons utiliser celui qui obtiendra les meilleures performances sur notre problème. C'est pourquoi nous testons un panel de six classifieurs. Parmi ces six méthodes de classification, cinq (*Multi-Layer Perceptron (MLP)*,*Random Forest (RF)*,*Stochastic Gradient Descent (SGD)*, *k-Nearest Neighbors (kNN)* et *Support Vector Machines (SVMs)*) ont été présentées en Section 4.4.3. Le dernier classifieur que nous allons évaluer est du type *Logistic Regression (LR)*. C'est un modèle de régression binomial utilisé pour décrire les relations entre une variable dépendante et une ou plusieurs variables indépendantes. Le modèle LR estime la probabilité de survenue d'une occurrence en utilisant une fonction sigmoïde.
Comme nous l'avons fait dans le Chapitre II.4, nous optimisons les hyper-paramètres qui contrôlent le processus d'apprentissage à l'aide de la méthode combinatoire *Grid-Search*. Cette méthode utilise un produit cartésien pour créer des tuples d'hyper-paramètres qui sont testés sur chaque classifieur. Le tuple retenu est celui offrant la meilleure performance (voir Section 4.4.3). Enfin, nous évaluons les six classifieurs en utilisant les hyper-paramètres optimisés et sélectionnons le meilleur.

## 6.4 Approche détaillée

Dans cette section, nous détaillons les différentes étapes présentées en Section 6.3. Nous commençons par expliciter la création du jeu de données. Nous détaillerons ensuite le traitement des différentes variables, le sur-échantillonage des données et enfin la sélection d'un classifieur.

### 6.4.1 Création d'un jeu de données annoté de développeurs

Comme présenté en Section 2.6, l'apprentissage supervisé consiste à apprendre à un algorithme la correspondance entre une entrée et une sortie sur la base d'exemples de paires entrée-sortie [RN20]. Ce type d'apprentissage requiert donc un jeu de données annotées pour pouvoir apprendre puis prédire un résultat en fonction d'une entrée. L'objectif poursuivi est la détection des ingénieurs informatique seniors sur la base de métriques logicielles. Nous distinguons deux catégories à classer de manière automatique : les développeurs dits "*senior software engineers*" et les autres, appelés "*non-senior software engineers*". Après recherche, il n'existe pas, à notre connaissance, de jeux de données de ce type. Nous avons donc pris la décision de le créer. La Figure II.6.1 présente le processus de création du jeu de données. Nous extrayons en parallèle le code source et les développeurs du dépôt GitHub puis calculons des métriques sur le code source, que nous associons

à chacun des développeurs. Nous obtenons ainsi un jeu de données non-annoté de développeurs et de leurs métriques. Nous annotons ensuite le jeu de données manuellement en exploitant diverses sources afin de qualifier le rôle du développeur dans le projet. De cette manière, nous obtenons un jeu de données étiqueté.

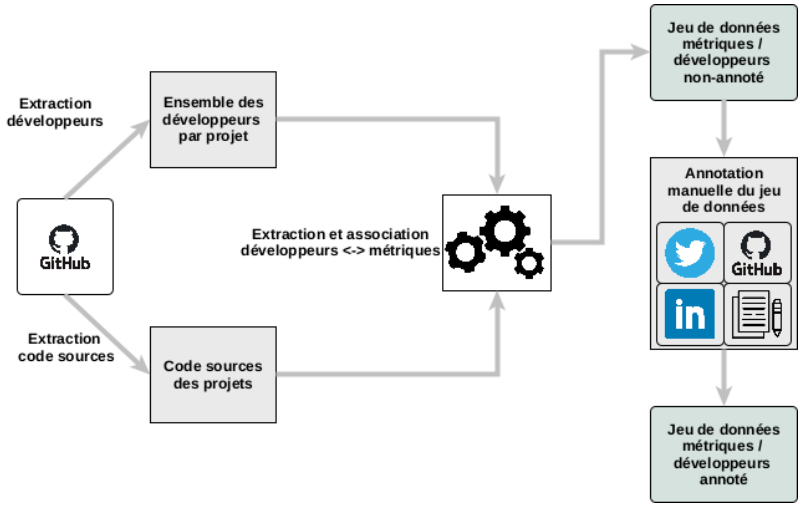



FIGURE II.6.1 – Processus de création du jeu de données de développeurs.

### 6.4.2 Extraction des développeurs

Nous souhaitons rechercher les développeurs seniors ayant un potentiel rôle architectural dans les projets. C'est pourquoi nous focalisons notre extraction sur des projets Java utilisant le *framework* architectural Spring présenté en Section 2.2.3. Ainsi, nous sélectionnons 17 projets sur GitHub qui disposent d'au moins 2 développeurs, avec au moins 2 versions et qui utilisent Spring. Sur la base de ces 17 projets, nous avons été en mesure d'extraire 951 développeurs. Les données des développeurs extraites au travers de l'API de GitHub concernent : le nom d'utilisateur (*username*), le nom et l'email de chacun des développeurs. Notons que chaque développeur est attaché à un seul et unique projet. Ainsi, une personne présente dans deux projets différents sera représentée comme deux développeurs distincts auxquels sont attachées des métriques différentes pour chacun des projets.

### 6.4.3 Extraction et association des métriques logicielles aux développeurs

Comme montré par la Figure II.6.1, en parallèle de l'extraction des développeurs, nous extrayons les codes sources du dernier *commit* connu des 17 projets qui composent le jeu de données. Sur ces codes sources nous utilisons l'outil PyDriller [1] [SAB18] que nous avons modifié afin d'extraire

1. https://github.com/ishepard/pydriller

23 métriques (appelées *features* en apprentissage machine). Nous présentons ces métriques dans la Table II.6.2. Ces métriques sont choisies pour leurs relations avec le design et l'architecture logicielle (`NoAB`, `NonAB`, `NoCII`, `NoCnII`, `NoCE`, `NonCE`, `NoInEI`, `NoIEI`), l'architecture *runtime* Spring (`AddSAM`, `DelSAM`, `ChurnSAM`) et la structure Gradle ou Maven (*`AddLGM`*,*`DelLGM`*,*`ChurnLGM`*,*`NoMGM`*). Di Bella *et al.* [DBSS13] utilisent neuf métriques pour regrouper les développeurs en quatre catégories en utilisant des méthodes d'apprentissage non-supervisées. Di Bella *et al.* ont montré que quatre métriques étaient discriminantes pour classer les développeurs : le nombre de *commits* (`NoC`), le nombre de lignes de code (`AddLoC`, `DelLoC`, `ChurnLoC`), le nombre de jours dans le projet (`DiP`) et enfin le temps moyen entre deux *commits* (`IT`). Afin de capitaliser sur la publication de Di Bella *et al.*, nous réutilisons ces métriques dans notre contexte de classification.
Nous portons une attention particulière aux développeurs n'ayant pas de contributions au niveau du code. Ceux-ci sont susceptibles d'ajouter du bruit aux données et ainsi réduire la qualité du classifieur. Nous supprimons donc du jeu de données les développeurs n'ayant pas modifié au moins une ligne de code. Cela revient à vérifier qu'au moins une des variables parmi `AddLoC`, `DelLoC`, `AddAM` et `DelAM` a une valeur non nulle. Ces suppressions font passer le nombre de développeurs de notre jeu de données de 951 à 703.

### 6.4.4 Annotation du jeu de données

Dans cette section, nous allons détailler l'annotation et le filtrage des valeurs aberrantes de notre jeu de données de développeurs.

#### 6.4.4.1 Annotation manuelle via les réseaux sociaux et la documentation

Comme montré par la Figure II.6.1, la dernière étape consiste à annoter le jeu de données. Pour cela nous associons les développeurs extraits de GitHub à leur rôle dans les projets. Nous avons cherché sur internet l'ensemble des développeurs à l'aide de leur nom et *username* GitHub afin de trouver l'information permettant d'attacher au développeur un rôle dans le projet. Nous utilisons cette méthode car Archambault et Grudin [AG12] ont montré que beaucoup de développeurs utilisent des réseaux sociaux. Une autre solution consisterait à envoyer des questionnaires aux développeurs. Cette dernière méthode présente une faiblesse majeure : le faible taux de réponse généralement obtenu [Tse98, CHT00]. C'est pourquoi nous adoptons une méthode d'annotation manuelle sur la base d'éléments trouvés sur les réseaux sociaux LinkedIn, Twitter, Facebook mais également sur les profils GitHub ou encore dans la documentation des projets. En considérant un projet donné, si les informations trouvées sur le développeur mentionnent :

- "*Architect*" ou "*Senior Software Engineer*" alors nous catégorisons le développeur comme "Senior Software Engineer" (SSE),
- "*Junior Software Engineer" ou "Software Engineer*" alors nous catégorisons ce développeur comme "Software Engineer" (SE),

| Acronyme | Variable | Description |
|---|---|---|
| NoAB | Nombre de classes abstraites | Nombre de classes abstraites créées par un développeur donné |
| NonAB | Nombre de classes non abstraites | Nombre de classes non abstraites créées par un développeur donné |
| NoCII | Nombre de classes implémentant une interface | Nombre de classes implémentant une interface créées par un développeur donné |
| NoCnII | Nombre de classes n'implémentant pas une interface | Nombre de classes n'implémentant pas d'interface créées par un développeur donné |
| NoCE | Nombre de classes étendant une autre classe | Nombre de classes utilisant l'héritage créées par un développeur donné |
| NonCE | Nombre de classes n'étendant pas une autre classe | Nombre de classes sans héritage créées par un développeur donné |
| NoInEI | Nombre d'interfaces n'étendant pas une autre interface | Nombre d'interfaces n'utilisant pas d'interface créées par un développeur donné |
| NoIEI | Nombre d'interfaces étendant une autre interface | Nombre d'interfaces utilisant une interface créées par un développeur donné |
| *AddLGM* | Lignes de code Gradle ou Maven ajoutées | Nombre de lignes de code Gradle ou Maven ajoutées par un développeur donné |
| *DelLGM* | Lignes de code Gradle ou Maven supprimées | Nombre de lignes de code Gradle ou Maven supprimées par un développeur donné |
| *ChurnLGM* | Différence entre les lignes de code Gradle ou Maven ajoutées et supprimées | Différence entre les lignes de code Gradle ou Maven ajoutées et supprimées par un développeur donné |
| *NoMGM* | Nombre de modules Gradle ou Maven | Nombre de modules Gradle ou Maven créés par un développeur donné |
| AddSAM | Lignes de code spécifiques à Spring ajoutées | Nombre de lignes de code Spring ajoutées par un développeur donné |
| DelSAM | Lignes de code Spring supprimées | Nombre de lignes de code Spring supprimées par un développeur donné |
| ChurnSAM | Différence entre les lignes de code Spring ajoutées et supprimées | Différence entre les lignes de code Spring ajoutées et supprimées par un développeur donné |
| AddLoC | Lignes de code ajoutées | Nombre de lignes de code Java ajoutées par un développeur donné |
| DelLoC | Lignes de code supprimées | Nombre de lignes de code Java supprimées par un développeur donné |
| ChurnLoC | Différence entre les lignes de code ajoutées et supprimées | Différence entre les lignes de code Java ajoutées et supprimées par un développeur donné |
| DiP | Nombre de jours dans le projet | Nombre de jours dans le projet pour un développeur donné (temps entre le premier et le dernier *commit*) |
| IT | Temps moyen inter-*commit* | Temps Nombre de jours moyen entre 2 *commits* pour un développeur donné |
| NoC | Nombre de *commits* | Nombre de *commits* effectués par un développeur donné. |
| AddF | Nombre de fichiers ajoutés | Nombre de fichiers ajoutés par un développeur donné. |
| DelF | Nombre de fichiers supprimés | Nombre de fichiers supprimés par un développeur donné. |

TABLE II.6.2 – Détail des 23 métriques extraites pour chaque développeur.

– "*Developer*", nous cherchons si le développeur fait mention d'un Master en Informatique (*Master of Sciences in Software Engineering*). Si oui alors le développeur est catégorisé "Software Engineer" (SE) sinon il est catégorisé "OTHER".

Si le nom d'utilisateur GitHub du développeur contient le mot BOT, il est étiqueté comme BOT. Enfin, si nous ne trouvons aucune information sur le rôle du développeur dans le projet, celui-ci est étiqueté UNKNOWN. Les étiquettes utilisées sont regroupées dans la Table II.6.3. Ces cinq étiquettes permettent des expérimentations plus complexes que la classification binaire (NSSE, SSE) que nous allons réaliser.

| **Acronyme de l'étiquette** | **Étiquette** | **Description** |
|---|---|---|
| SSE | Senior Software Engineer | Développeur déclarant être un ingénieur en informatique senior. |
| SE | Software Engineer | Développeur déclarant être un ingénieur en informatique. |
| OTHER | Other | Développeur déclarant ne pas appartenir à une des catégories ci-dessus. |
| UNKNOWN | Unknown | Pas d'information disponible à propos du développeur. |
| BOT | Bot | Machine d'intégration continue ayant un profil GitHub pour les phases de test, *commit*, versionnement, etc. |

TABLE II.6.3 – Étiquettes utilisées pour annoter le jeu de données.

#### 6.4.4.2 Filtrage et ré-annotation des aberrations

Comme nous l'avons évoqué plus haut, la recherche d'informations sur les rôles des développeurs est faite en utilisant des sources d'informations variées : LinkedIn, Twitter, Facebook, GitHub [AG12] ou la documentation des projets. Un des risques d'utiliser de manière exclusive cette méthode d'annotation vient du fait que les développeurs étiquetés SE ou UNKWOWN peuvent avoir des niveaux de métriques comparables à ceux des SSE. En effet, travaillant à partir d'informations déclaratives produites par les développeurs, il est possible que certains ne se déclarent pas SSE et inversement, ou qu'aucune description du rôle ne puisse être trouvée. De ce mauvais étiquetage peuvent découler des problèmes de classification des développeurs. Pour éviter ces erreurs d'étiquetage liées à une mauvaise identification des développeurs, nous avons ré-étiqueté manuellement certains développeurs. Cette étape de ré-annotation se découpe en deux phases :

1. Utiliser une méthode mathématique afin de détecter les valeurs aberrantes parmi l'ensemble des catégories de développeurs. Pour cela, nous utilisons la méthode dite de la forêt d'isolement. Celle-ci calcule un score d'aberration pour chacune des observations dans le jeu de données. Ce score est une mesure la "normalité" ou non de chaque observation par rapport à l'ensemble des observations du jeu de données. Nous faisons l'hypothèse que certains développeurs SE ou UNKWON sont des développeurs SSE non déclarés. Nous utilisons cette méthode afin de minimiser les biais que nous pourrions induire par un examen manuel de l'ensemble des 23 variables pour chaque développeur.

2. Après inspection des valeurs aberrantes renvoyées par la forêt d'isolement, nous ré-étiquetons manuellement 17 développeurs considérés comme valeurs aberrantes dans leurs catégories : 4 UNKNOWN deviennent SSE et 13 SE deviennent SSE. La Table II.6.4 donne le détail de cette ré-annotation par catégorie de développeurs.

Bien qu'ayant créé un jeu de données comportant cinq étiquettes, nous réduisons notre classification à un jeu de données binaire : Senior Software Engineer (SSE) et Non-Senior Software Engineer (NSSE). Nous faisons ce choix car il est plus simple dans un premier temps d'effectuer une classification binaire avant de l'étendre à plus de classes. De plus, comme le montre la Table II.6.4, nous avons parfois très peu d'observations dans certaines classes (comme BOT et OTHER) ce qui rend compliqué l'apprentissage d'un algorithme ou la création de données synthétiques. Ainsi, l'ensemble des développeurs ayant une étiquette autre que SSE sont ré-étiquetés NSSE. À l'issue de cette opération, nous obtenons 98 développeurs étiquetés SSE et 605 NSSE.

| | **#SSE** | **#SE** | **#UNKNOWN** | **#BOT** | **#OTHER** |
|---|---|---|---|---|---|
| **Avant ré-annotation manuelle** | 81 | 86 | 509 | 10 | 17 |
| **Après ré-annotation manuelle** | 98 | 73 | 505 | 10 | 17 |

TABLE II.6.4 – Développeurs dans chaque catégorie avant et après ré-annotation.

### 6.4.5 Traitement des variables

Pour être efficace, la classification requiert une phase de pré-traitement [ZC18] souvent appelée *feature engineering*. Dans cette phase, une variété de transformations est appliquée sur les données : mise à l'échelle, transformations mathématiques, normalisation, détection de valeurs aberrantes, etc..

La première étape du traitement des variables que nous appliquons est une transformation logarithmique pour réduire la dispersion des valeurs sur 11 des variables : `SAddDelLOC`, `DiP`, `NoC`, `SAddDelF`, `SAddDelSAM`, `SAddDelSM`, `AddLOC`, `DelLOC`, `AddSAM`, `DelSAM`, `AddSM` et `DelSM`. En effet, comme le montre la Table II.6.5, ces variables présentent des écarts très importants entre leur valeur minimale et leur valeur maximale. La moyenne est également très largement supérieure à la médiane sur l'ensemble des valeurs, montrant ainsi le caractère très dispersé des valeurs. Ces dispersions entraînent de grandes variances dans les estimations et, *in fine*, font décroître les performances des classifieurs.

Un autre point important à traiter concerne les différences d'échelles et d'unités des variables. Ces différences d'échelles peuvent amener à de moindres performances des classifieurs. C'est pourquoi nous appliquons une mise à l'échelle de nos variables. Pour cela, nous utilisons la méthode Min-Max afin de placer l'ensemble des valeurs dans l'intervalle $[-1, 1]$. La méthode Min-Max est définie comme suit :

$$X_{scaled} = \left( \frac{X - X_{min}}{X_{max} - X_{min}} \right) \times (max - min) + min$$

avec :

- $X$ la variable à mettre à l'échelle
- $X_{scaled}$ la variable mise à l'échelle
- $X_{min}$ et $X_{max}$ les valeurs minimum et maximum observées pour la variable $X$
- $max$ la borne supérieure de l'intervalle de mise à l'échelle
- $min$ la borne inférieure de l'intervalle de mise à l'échelle

### 6.4.6 Sur-échantillonage des données

Un point d'attention dans notre approche est le fait que notre jeu de données est fortement déséquilibré. Celui-ci comporte 98 développeurs étiquetés comme ingénieurs informatique seniors (SSE) contre 605 développeurs non ingénieurs informatique seniors (NSSE). Conserver ce déséquilibre dans les données peut conduire à fortement biaiser le modèle de classification. Pour surmonter ce manque de données dans la classe minoritaire (SSE), nous utilisons la méthode de sur-échantillonage SMOTE (*Synthetic Minority Over-sampling Technique*) [CBHK02]. SMOTE a été conçue pour créer des données synthétiques en se basant sur les données existantes. SMOTE sélectionne aléatoirement des points existants dans un espace, trace des vecteurs entre ces points puis génère

| | Variable | | | | | | |
|---|---|---|---|---|---|---|---|
| | DiP | NoC | SAddDelLOC | AddLOC | DelLOC | SAddDelF | SAddDelSAM |
| Maximum | 6775 | 4094 | 1557349 | 1328791 | 228558 | 19523 | 20528 |
| Minimum | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| Écart-type | 948 | 362 | 72557 | 61262 | 14010 | 1032 | 814 |
| Moyenne | 429 | 91 | 9328 | 7836 | 1491 | 126 | 67 |
| 1er quartile | 1 | 1 | 8 | 5 | 1 | 0 | 0 |
| Médiane | 3 | 3 | 58 | 43 | 5 | 0 | 0 |
| 3ème quartile | 382 | 13 | 513 | 372 | 76 | 4 | 0 |

| | Variable | | | | |
|---|---|---|---|---|---|
| | **AddSAM** | **DelSAM** | **SAddDelSM** | **AddSM** | **DelSM** |
| Maximum | 11898 | 8630 | 23028 | 12058 | 10970 |
| Minimum | 0 | 0 | 0 | 0 | 0 |
| Écart-type | 468 | 353 | 1141 | 573 | 622 |
| Moyenne | 39 | 28 | 145 | 77 | 68 |
| 1er quartile | 0 | 0 | 0 | 0 | 0 |
| Médiane | 0 | 0 | 0 | 0 | 0 |
| 3ème quartile | 0 | 0 | 4 | 2 | 0 |

TABLE II.6.5 – Description statistique des 11 variables où la transformation logarithmique est appliquée.

aléatoirement de nouveaux points (données synthétiques) sur les vecteurs. Cette méthode a montré ses performances en comparaison d'autres méthodes de sur-échantillonnage comme la génération de données aléatoires ou la copie de données [CBHK02, DM20]. Ici, nous sur-échantillonnons avec une variante de SMOTE, appelée $k$-Means SMOTE [LDB17], pour générer des profils de développeurs seniors (SSE). Par rapport à SMOTE, $k$-Means SMOTE réduit le bruit dans les données synthétiques générées. La génération de données synthétiques est effectuée durant la phase d'entraînement du classifieur afin d'augmenter le nombre de données utilisées pour l'apprentissage. La Figure II.6.2 montre la méthode de sur-échantillonnage combinée à l'évaluation *4-fold* stratifiée. Pour chaque ensemble d'entraînement dans les *folds*, nous appliquons un sur-échantillonnage de la classe minoritaire avec la méthode $k$-means SMOTE. Ainsi après sur-échantillonnage sur chaque *fold*, la taille de la classe minoritaire devient la même que celle de la classe majoritaire. Nous ne faisons pas de sur-échantillonnage sur la globalité du jeu de données avant d'entraîner le classifieur. En effet, procéder de la sorte permettrait d'équilibrer le jeu de données mais les données utilisées pour l'évaluation du classifieur comporteraient des données synthétiques. Or, pour que l'évaluation soit pertinente, nous souhaitons que celle-ci soit faite sur des données réelles. Comme le montre la Figure II.6.2, la version stratifiée de *k-fold* permet de conserver la distribution originale des données dans chaque classe (SSE et NSSE) pour chaque *fold*. Nous choisissons une valeur de $k$ petite ($k = 4$) et une version stratifiée à cause de la faible proportion de SSE dans le jeu de données. Après validation croisée, nous comparons les résultats obtenus par les six classifieurs et retenons celui ayant la meilleure performance en termes de mesure F1.

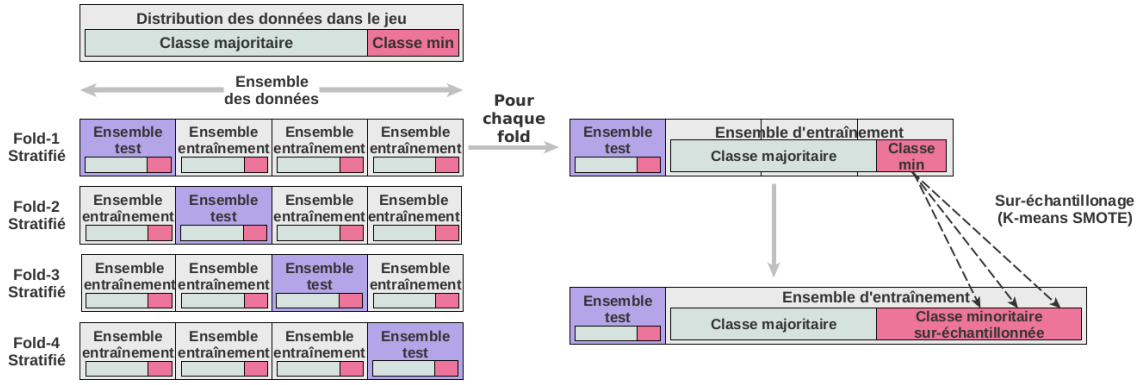



FIGURE II.6.2 – Génération de données synthétiques durant l'évaluation 4-*fold*.

### 6.4.7 Sélection d'un classifieur

Dans ces travaux, nous comparons six classifieurs différents. Nous testons trois classifieurs ayant de bonnes performances sur les petits jeux de données (LR, SVMs, $k$-NN) et, à titre de comparaison, trois classifieur plus complexes (SGD, RF, MLP). La Figure II.6.3 présente les six classifieurs évalués. Chacun de ces classifieurs voit ses hyper-paramètres optimisés par l'algorithme *Grid-Search* que nous avons décrit en Section 4.4.3. Les performances des classifieurs sont évaluées en

utilisant une validation croisée *k-fold* ($k = 4$) dans sa version stratifiée (voir Figure II.6.2) et sont comparées en fonction de la valeur de leur mesure F1.

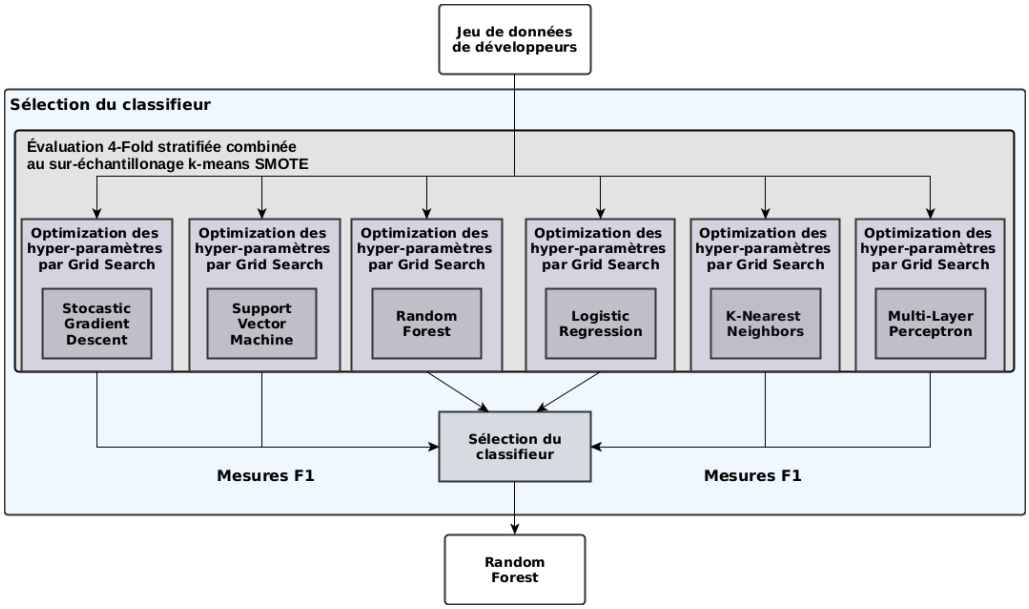



FIGURE II.6.3 – Processus de sélection du classifieur pour la classification de développeurs.

## 6.5 Résultats

Dans cette section, nous présentons les résultats obtenus par les différents classifieurs, nous comparerons ensuite différentes configurations du meilleur classifieur. Enfin, nous utiliserons une méthode d'explicabilité pour comprendre les mécanismes de classification liés au classifieur.

### 6.5.1 Sélection d'un classifieur

Comme nous l'avons vu dans la Section 6.4.7, nous évaluons six classifieurs : *Logistic Regression* (LR), *k-Nearest Neighbors* (*k*NN), *Stochastic Gradient Descent* (SGD), *Support Vector Machines* (SVMs), *Random Forest* (RF) et *Multi-Layer Perceptron* (MLP). Ces classifieurs ont vu leurs hyper-paramètres optimisés à l'aide de l'algorithme *Grid-Search*. Cette évaluation est faite en utilisant une évaluation 4-*fold* stratifiée puis en calculant la moyenne des mesures F1 obtenues sur l'ensemble des *folds*. La Table II.6.6 présente les résultats obtenus. Les ensembles de paramètres trouvés par *Grid-Search* sont aussi donnés dans la Table II.6.6. En utilisant ces hyper-paramètres optimisés, le classifieur *Random Forest* donne les meilleurs résultats (0.782). Nous constatons également que l'ensemble des classifieurs testés obtiennent des performances correctes sur la classification de notre jeu de données, la plus mauvaise performance constatée étant celle du classifieur *Logistic Regression* (0.757).

| **Classifieur** | **Hyper-paramètres optimisés par Grid-Search** | **Mesure F1** | **Protocole d'évaluation** |
|---|---|---|---|
| **RF** | criterion='gini'<br>n_estimators=300<br>random_state=0<br>max_depth=2<br>max_features='log2' | **0.7887**<br>**IC 95%: 0.053** | 4-*fold*<br>Stratifiée |
| SGD | loss='modified_huber'<br>max_iter=2000<br>random_state=0<br>tol=0.1<br>alpha=0.1<br>learning_rate='invscaling' | 0.775<br>IC 95%: 0.057 | 4-*fold*<br>Stratifiée |
| SVM | C=0.2<br>gamma='scale'<br>kernel='poly'<br>random_state=0<br>tol=0.0001 | 0.763<br>IC 95%: 0.021 | 4-*fold*<br>Stratifiée |
| kNN | weights='distance'<br>n_neighbors=6<br>algorithm='ball_tree'<br>p=2 | 0.767<br>IC 95%: 0.073 | 4-*fold*<br>Stratifiée |
| MLP | activation='relu'<br>learning_rate='constant'<br>max_iter=100<br>random_state=0<br>hidden_layer_sizes=(50, 50)<br>solver='adam' | 0.766<br>IC 95%: 0.046 | 4-*fold*<br>Stratifiée |
| LR | C=0.52<br>random_state=9090<br>solver='sag'<br>tol=0.1 | 0.757<br>IC 95%: 0.142 | 4-*fold*<br>Stratifiée |

TABLE II.6.6 – Résultats obtenus en utilisant les classifieurs issus de Scikit-learn avec évaluation 4-*fold* stratifiée.

### 6.5.2 Résultats intermédiaires

Nous avons évalué l'influence des différentes étapes liées au traitement des donnés, à la génération de données synthétiques et à l'optimisation du classifieur (ici *Random Forest*). Pour cela, nous définissons 4 configurations où nous calculons 4 métriques : mesure F1, rappel, précision et justesse en utilisant une évaluation 4-*fold* stratifiée. La Table II.6.6 et la Figure II.6.4 présentent les résultats des 4 configurations testées.

La configuration 1 utilise le classifieur *Random Forest* avec le paramétrage par défaut de la bibliothèque Scikit-learn. Dans cette configuration la transformation logarithmique, la mise à l'échelle des données ainsi que sur-échantillonnage ne sont pas mis en œuvre. Cette configuration donne des résultats corrects en termes de mesure F1 (0.7601) malgré un classifieur paramétré par défaut et aucune transformation sur les variables.

La configuration 2 est identique à la 1 à l'exception des valeurs des hyper-paramètres qui sont optimisés à l'aide de *Grid-Search*. La mesure F1 n'est quasiment pas affectée (0.7591) par rapport à la configuration précédente (0.7601). Les intervalles de confiance sont en revanche réduits par rapport à la configuration 1, ce qui montre que l'utilisation de bons hyper-paramètres permet d'avoir un classifieur plus précis.

La configuration 3 utilise la transformation logarithmique ainsi que la mise à l'échelle des variables couplées au classifieur de la configuration 2. Dans cette configuration, le sur-échantillonage n'est pas effectif. Cette configuration améliore légèrement la performance sur la mesure F1 (0.7654) et le rappel (0.7033) mais a le défaut d'augmenter les intervalles de confiance sur l'ensemble des mesures.

La configuration 4 utilise la configuration 3 en ajoutant le sur-échantillonage via $k$-Means SMOTE. Le sur-échantillonage a un effet d'amélioration clair sur la mesure F1 (0.778), le rappel (0.7446), la précision (0.8390), la justesse (0.8607) ainsi que sur l'ensemble des intervalles de confiance.

Ces différentes configurations montrent que les différentes étapes que nous mettons en place ont une influence positive sur les résultats de la classification. La mise en place du traitement des variables ainsi que la génération de données synthétiques permettent un réel gain de performance entre la configuration 1 (mesure F1 de 0.7601) et la configuration 4 (mesure F1 de 0.778).

| | **Configuration 1** | | **Configuration 2** | | **Configuration 3** | | **Configuration 4** | | **Protocole d'évaluation** |
|---|---|---|---|---|---|---|---|---|---|
| | **Valeur** | **IC 95%** | **Valeur** | **IC 95%** | **Valeur** | **IC 95%** | **Valeur** | **IC 95%** | |
| **F1** | 0.7601 | 0.0571 | 0.7591 | 0.0458 | 0.7654 | 0.1111 | 0.7887 | 0.0529 | Stratified 4-fold |
| **Rappel** | 0.6829 | 0.0978 | 0.6933 | 0.0777 | 0.7033 | 0.1267 | 0.7446 | 0.0708 | Stratified 4-fold |
| **Précision** | 0.8608 | 0.0625 | 0.8412 | 0.0654 | 0.8411 | 0.1098 | 0.8390 | 0.0438 | Stratified 4-fold |
| **Justesse** | 0.8324 | 0.0459 | 0.8359 | 0.0363 | 0.8409 | 0.0678 | 0.8607 | 0.0364 | Stratified 4-fold |

TABLE II.6.7 – Valeurs et intervalles de confiance pour les mesures de F1, rappel, précision et justesse sur 4 configurations.

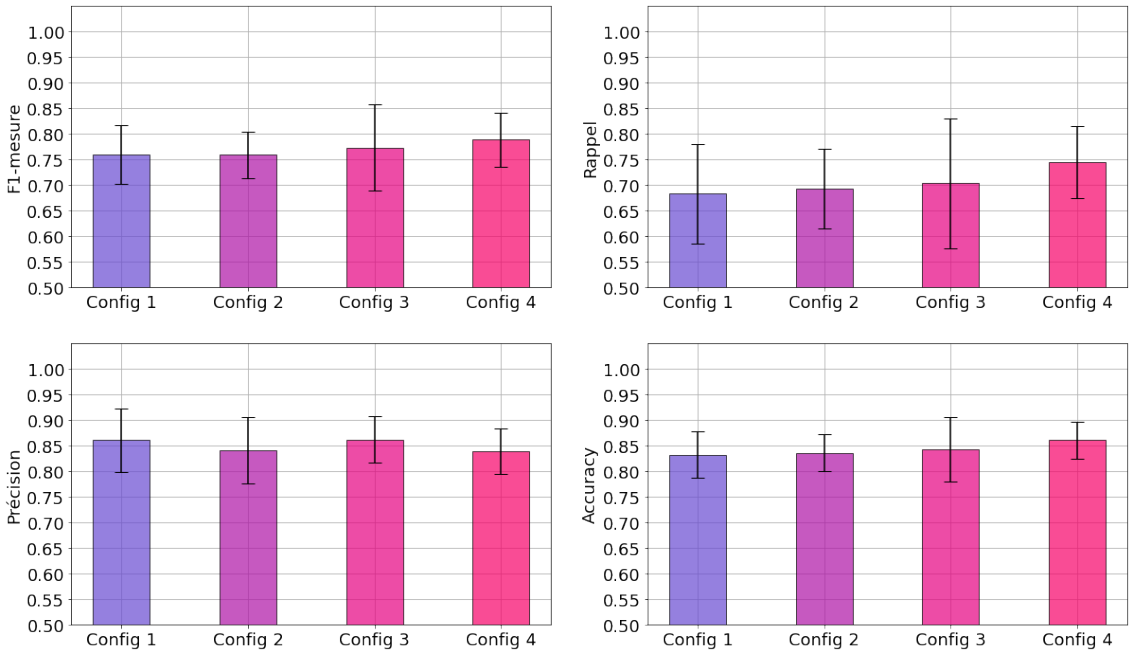



FIGURE II.6.4 – Représentation graphique des valeurs de la Table II.6.6.

### 6.5.3 Explicabilité du classifieur

Nous l'avons vu dans la Section 4.6 que l'explicabilité d'un classifieur peut être réalisée de manière locale (explication de la classification d'une ou plusieurs instances) mais aussi de manière globale (explication des mécanismes internes au classifieur). Ici, nous faisons une explication globale de *Random Forest* car nous souhaitons explorer les variables sur lesquelles s'appuie le classifieur pour prendre ses décisions. Comme nous l'avons mentionné en Section 4.6, deux principales méthodes agnostiques d'explicabilité existent : la permutation de variables et la méthode SHapley Additive exPlanations (SHAP) [LL17]. Ces deux méthodes permettent d'évaluer l'importance des variables pour le classifieur. La permutation de variables est simple mais s'avère sensible aux variables corrélées. Les corrélations de variables ont tendance à biaiser l'importance de celles-ci. Ici, nous utilisons 23 variables (voir Table II.6.2) dont certaines sont fortement corrélées. La Figure II.6.5 présente les coefficients calculés par la corrélation de Spearman entre chacune des 23 variables. Cette figure montre des valeurs de corrélation très importantes pour certaines variables. Nous pouvons par exemple citer la corrélation de 0.85 entre le nombre de *commits* (`NoC`) et le nombre de jours dans le projet (`DiP`) ou encore le coefficient de 0.80 entre le nombre de fichiers ajoutés (`AddF`) et le nombre de lignes de code ajoutées (`AddLOC`). Ces valeurs de corrélation élevées justifient l'utilisation de SHAP comme méthode d'explicabilité du classifieur du fait de sa moindre sensibilité aux variables corrélées.

En utilisant SHAP, nous avons calculé l'importance des 23 variables afin de mieux comprendre le fonctionnement de notre classifieur. Pour assurer la confiance et la précision des résultats, nous

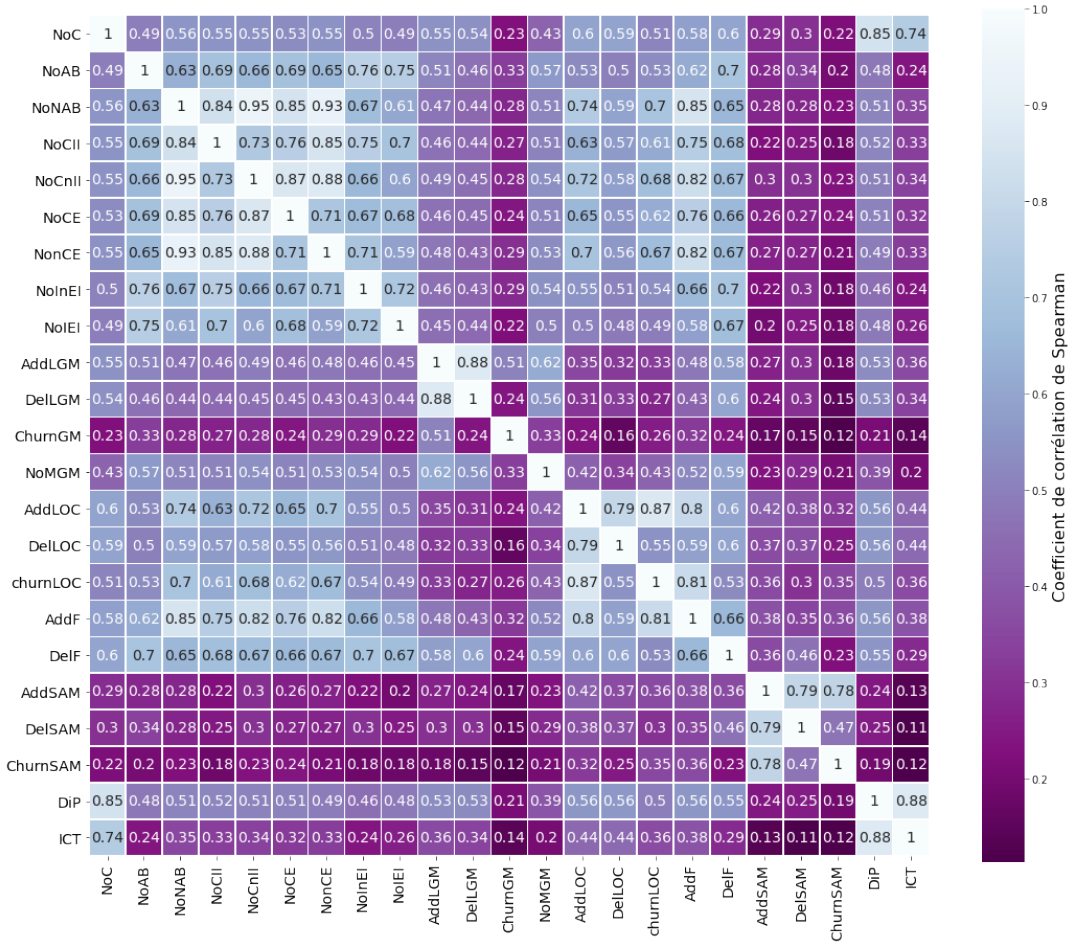
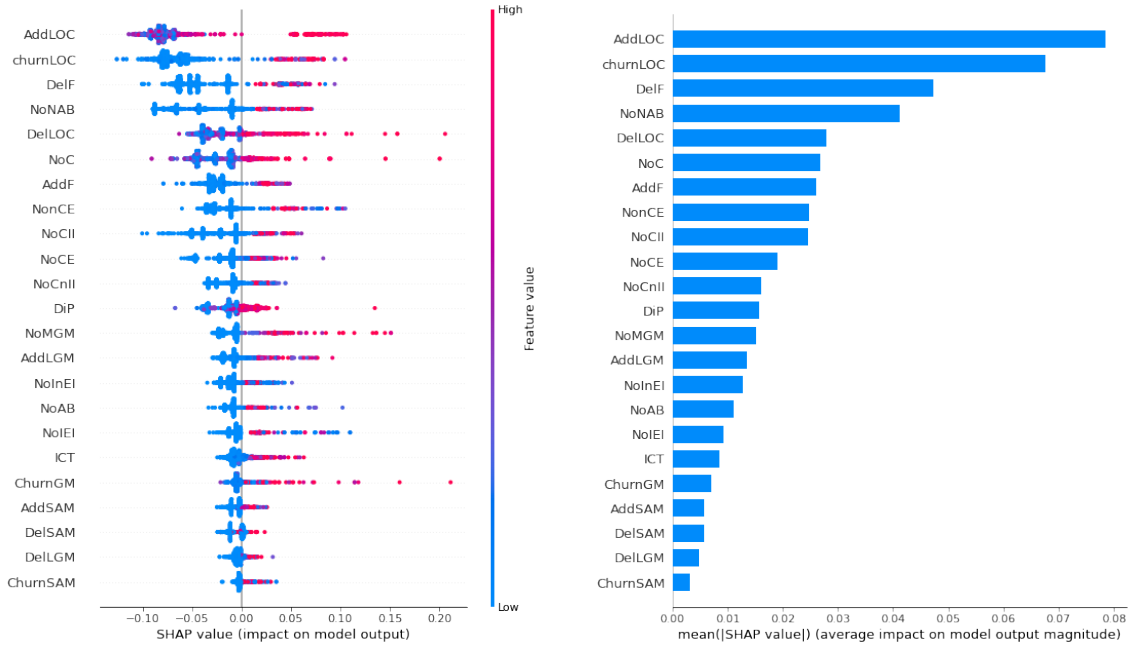



FIGURE II.6.5 – Matrice de corrélation (Spearman) des 23 variables décrites dans la Table II.6.2.

avons utilisé SHAP sur chaque *fold* puis nous avons moyenné les résultats. Cette méthode permet d'avoir une meilleure représentativé de l'impact des variables par rapport à un calcul réalisé sans découpage sur l'ensemble du jeu de données, d'autant plus que nous faisons de la génération de données synthétiques lors des différents *folds*. Les résultats de l'application de SHAP sur les quatre *folds* sont donnés dans la Figure II.6.6. Nous constatons que quatre variables ont une importance prépondérante sur la classification d'un développeur : le nombre de lignes de code ajoutées (`AddLOC`), le churn sur les lignes de code (`ChurnLOC`), le nombre de fichiers supprimés (`DelF`) et le nombre de classes non-abstraites créées (NoNAB). Di Bella *et al.* [DBSS13] ont montré que le nombre de lignes de code était une variable discriminante pour la catégorisation des développeurs à l'aide d'une méthode non-supervisée. Nous faisons ici le même constat avec une classification supervisée. Le nombre de fichiers supprimés ainsi que le nombre de classes non-abstraites sont l'indice d'actions de *refactoring* réalisées par les développeurs expérimentés. Parmi les variables ayant une importance moindre, le nombre de *commits* (`NoC`) est en sixième position et le nombre de




(a) Valeurs de SHAP obtenues pour chacune des instances classifiées.

(b) Moyennes absolues des valeurs de SHAP obtenues pour chacune des instances classifiées.

FIGURE II.6.6 – Valeurs de SHAP obtenues pour les 23 variables sur 4 *folds*.

jours dans le projet en douzième position. Ces variables sont également jugées discriminantes par Di Bella *et al.* Figurent ensuite différentes variables liées à la structure Java du projet : le nombre de classes n'étendant pas autre classe (`NonCE`), le nombre de classes implémentant (`NoCII`) ou non (`NoCnII`) une interface. Ces trois variables tendent à montrer que tenir compte de la structure Java du projet fait partie des facteurs qui permettent de discriminer les développeurs expérimentés des autres. Le nombre de modules Gradle ou Maven créés (`NoMGM`) et le nombre de lignes Gradle ou Maven ajoutées (`AddLGM`) ont une importance moins forte que la structure Java mais ne sont pas à négliger pour autant. Ces deux variables se placent en treizième et quatorzième position. Le constat le plus surprenant que nous faisons est celui des variables liées à l'architecture Spring (`AddSAM`, `DelSAM`, `ChurnSAM`), qui sont celles ayant le moins d'importance sur la classification. Les résultats obtenus dans le Chapitre II.5 pouvaient nous laisser penser que ces variables auraient plus de poids sur la classification des développeurs.

## 6.6 Synthèse, limites et perspectives

Dans ce chapitre, nous avons présenté une approche permettant de classer les développeurs expérimentés en fonction de 23 métriques logicielles. En premier lieu, nous avons construit un jeu de données annotées de développeurs. Puis, nous avons généré des données synthétiques afin d'équilibrer

le jeu de données et effectué la sélection d'un classifieur. Enfin, nous avons évalué les performances de notre classifieur et fait un travail d'explicabilité globale de celui-ci.

### 6.6.1 Synthèse

**Synthèse Question de recherche 6.1**

Les résultats que nous obtenons à l'aide de notre approche de classification des développeurs montrent que nous pouvons répondre positivement à la Question de recherche 6.1. Notre classifieur (*Random Forest*) dispose d'une bonne mesure F1 (0.7887) et d'une bonne précision (0.8390), le rappel est plus faible (0.7446). Ces résultat montrent que notre classifieur est capable d'être très précis sur les personnes profilées comme étant des développeurs expérimentés. En revanche, la capacité du classifieur à sélectionner des développeurs expérimentés est plus limitée. À notre sens, cela n'est pas un défaut car, pour notre usage, il est préférable de sélectionner précisément un nombre restreint de développeurs plutôt qu'un plus grand nombre avec moins de précision. En effet, comme nous l'ont montré Di Bella *et al.* [DBSS13] un projet logiciel contient un petit noyau de développeurs très expérimentés et c'est ces développeurs que nous ciblons avec notre classifieur.

**Synthèse Question de recherche 6.2**

Afin de répondre à la Question de recherche 6.2, nous avons étudié l'explicabilité globale de notre classifieur à l'aide de la méthode *k-means* SMOTE [LDB17]. L'explicabilité de notre classifieur montre que, dans le cas de notre classifieur (*Random Forest*), les métriques orientant le plus la classification des développeurs sont le nombre de lignes de code ainsi que le churn des lignes code. Contrairement à ce que nous aurions pensé, les métriques liées à l'architecture *runtime* n'influencent que très peu la décision du classifieur. Le nombre de fichiers ajoutés, le nombre de *commits* ainsi que les variables liées à la structure Java figurent parmi celles ayant une influence moyenne. Cela pourrait signifier indirectement que les développeurs expérimentés font beaucoup de modifications de la structure du projet.

### 6.6.2 Incertitudes sur la validité

#### 6.6.2.1 Validité externe

Les résultats que nous obtenons ici ont été produits sur 16 projets partageant des caractéristiques communes. Ces projets sont tous écrits dans le langage Java et utilisent le *framework* Spring. Ainsi, les profils de développeurs que nous obtenons pourraientt être spécifiques aux technologies utilisées. Il serait nécessaire de généraliser en disposant de profils de développeurs issus de projets écrits dans d'autres langages (C++, Javascript, etc.) et/ou d'autres types de *frameworks* d'architecture comme

Apache Struts, OSGI ou encore JEE. De plus, l'ensemble des développeurs extraits proviennent de projets logiciels *open-source* sur GitHub. Il se peut que, dans des projets industriels non *open-source*, les développeurs présentent des profils différents.

#### 6.6.2.2 Validité interne

Les étiquettes du jeu de données associées aux développeurs ont été ajoutées sur la base de déclarations faites sur les réseaux sociaux (LinkedIn, Twitter) ou dans la documentation des projets. Cet étiquetage déclaratif peut être l'objet d'un biais si certains développeurs se considèrent comme seniors alors qu'ils ne le sont pas ou inversement. Cela est aussi lié à la dimension subjective de la notion d'expérience en développement logiciel (voir Section 2.4). Bien qu'ayant minimisé ce risque en ré-étiquetant les développeurs à l'aide d'une méthode mathématique, des erreurs peuvent encore exister.

### 6.6.3 Limites

Dans le cadre de ces travaux, une première limite concerne les variables choisies dans le jeu de données. Il est possible que celles-ci soient un facteur limitant de la performance du classifieur. D'autres types de métriques liées au code ou au processus existent. Ainsi, il possible qu'en utilisant plus de métriques ou en remplaçant certaines de nos métriques par d'autres nous obtenions de meilleurs résultats. Un deuxième point d'attention provient de l'annotation du jeu de données : celui-ci est annoté sur la base de déclarations des développeurs. Or, il se peut que celles-ci ne soient pas à jour ou soient mal renseignées et viennent, *in fine*, fausser la classification. Une troisième limitation est le manque de généricité de l'approche. Nous utilisons des variables liées à des technologies (Maven, Gradle, Spring) et la solution proposée n'est pas agnostique à ces technologies.

### 6.6.4 Manipulation de données et éthique

Dans ce chapitre, nous présentons une méthode de classification automatique des développeurs. Nous sommes amenés dans ce cadre à manipuler des données personnelles (*e-mails*, noms et prénoms) et sommes donc contraints à l'anonymisation des données conformément à la Réglementation Générale sur la Protection des Données personnelles (RGPD) [2]. De plus, travailler avec des données produites par des humains et classer les-dits humains sur la base de celles-ci pose de nombreux problèmes éthiques. Ce système pourrait en effet être utilisé pour décider du statut social d'une personne. Il est aisé de comprendre qu'un développeur expérimenté aura potentiellement une considération (humaine, financière, etc.) plus grande qu'une personne inexpérimentée. Utiliser une classification automatique pour la gestion de ressources humaines et la hiérarchisation des développeurs est un usage que nous déconseillons très fortement. Comme pour toute technologie, l'usage doit en être raisonné et rester dans des limites sociales, morales et éthiques convenues.

2. https://www.cnil.fr/fr/reglement-europeen-protection-donnees

### 6.6.5 Perspectives

Les travaux présentés dans ce chapitre ouvrent de nombreuses perspectives.
Une première perspective concerne l'augmentation de la généricité du classifieur. Nous avons vu que les variables liées à Spring n'avaient que très peu d'influence sur la classification. Il serait donc possible de supprimer ces variables afin de pouvoir travailler avec le classifieur sur un panel plus large de projets Java.
Une deuxième perspective en lien avec la précédente concerne l'étude d'autres types de projets utilisant le même langage (Java) ou en utilisant d'autres. Un exemple serait l'adaptation de ce travail à C++. Il serait possible de reprendre les variables concernant la structure objet, de supprimer les variables spécifiques à Spring et de remplacer le dénombrement d'élément portant sur Spring ou Gradle par ceux présents dans un Makefile.
Une troisième perspective concerne l'amélioration et l'enrichissement du jeu de données en vue d'obtenir de meilleures performances. Nous travaillons ici sur des projets *open-source* mais il serait possible d'enrichir le jeu de données avec des projets en sources fermées provenant d'industriels.

# Chapitre II.7

# Étude des profils de développeurs expérimentés par croisement de données et apprentissage machine

**Sommaire**

Nous avons vu lors de la présentation du contexte de notre étude (voir Section 2.4) que les compétences des architectes sont plurielles. Cette pluralité peut s'exprimer sous différents aspects et notamment sous un angle technique. L'expertise technique et technologique est le fruit d'une évolution du développeur dans le projet. C'est précisément cette évolution sur un plan qualitatif que nous allons questionner dans ce chapitre. Pour cela, nous allons réutiliser les propositions faites aux Chapitres II.5 et II.6.

## 7.1 Questions de recherche

Dans le Chapitre II.5, nous avons bâti un modèle de mesure de la contribution logicielle avec lequel nous pouvons catégoriser les développeurs. Ainsi, de manière purement statistique, nous avons classés les développeurs en fonction de leur niveau de contribution à l'architecture. Dans le Chapitre II.6, nous avons utilisé l'apprentissage automatique pour catégoriser les développeurs expérimentés et non-expérimentés. Cette proposition repose sur un classifieur supervisé et entraîné sur un jeu de données de 703 développeurs décrits chacun par 23 variables. Ces deux approches étant différentes, nous allons les confronter. Cela nous inspire deux questions de recherche.

**Question de recherche 7.1**

Comment sont répartis les développeurs expérimentés en terme de contributions à l'architecture ?

La Question de recherche 7.1 croise l'approche de classification automatique des développeurs et celle permettant de catégoriser les développeurs en fonction de leur taux de contribution architectural afin d'étudier les types de contributeurs composant les développeurs expérimentés. Elle permettra ainsi de questionner la pertinence de la classification des développeurs expérimentés dans la gestion de préoccupations architecturales (triage de tickets, *turn-over* par exemple).
Un deuxième aspect que nous questionnons est celui de la transition du statut de développeur non-expérimenté à celui de développeur expérimenté.

**Question de recherche 7.2**

Quelle est l'évolution des caractéristiques permettant la transition du statut de développeur non-expérimenté à celui d'expérimenté ?

La Question de recherche 7.2 s'intéresse au changement de catégorie des développeurs. À travers cette question, nous faisons une analyse qualitative des variables qui permettent l'évolution des développeurs d'une catégorie à une autre. Pour ce faire, nous utilisons les travaux sur la classification de développeurs, décrits au Chapitre II.6, dans une nouvelle perspective.

## 7.2 Méthodologie

Dans cette section, nous décrivons le cas d'étude analysé ainsi que la méthodologie suivie pour répondre à ces deux hypothèses.

### 7.2.1 Cas d'étude *BroadleafCommerce*

Comme au Chapitre II.5, nous focalisons notre étude sur le projet *open-source BroadleafCommerce*. Ce projet a été précédemment sélectionné et décrit (Section 5.4). Nous étudions ici aussi les 13 versions majeures et mineures du projet. Disposant de données déjà extraites, nous sommes en mesure de les croiser avec de nouvelles données.

### 7.2.2 Approche proposée

La réponse à notre Question de recherche 7.1 nécessite le croisement de données issues des propositions décrites dans les Chapitres II.5 et II.6. Ainsi, le traitement appliqué à chaque version du projet étudié se scinde en deux processus parallèles et se termine par une phase de croisement des données :

- Un premier processus permet la classification automatique des développeurs en deux catégories (expérimentés ou non) à l'aide du classifieur créé au Chapitre II.6. Ce processus est décrit par la Figure II.7.1. Après avoir obtenu les développeurs participant au projet, nous extrayons pour chacun les 23 variables nécessaires à leur classification. Puis, à l'aide du jeu de données ainsi créé, nous utilisons le classifieur afin de classer de manière automatisée les développeurs. Nous obtenons un jeu de données de développeurs classés de manière automatique par le classifieur.

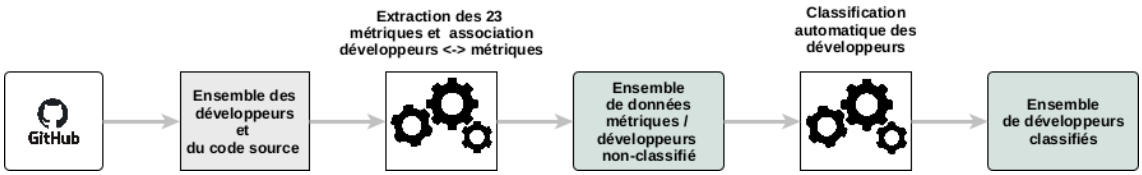



FIGURE II.7.1 – Processus de classification des développeurs, sur la base des travaux du Chapitre II.6

- En parallèle, un second processus, décrit par la Figure II.7.2, calcule les taux de contribution (architecturaux ou non) de chacun des développeurs en vue de leur catégorisation statistique comme décrit au Chapitre II.5.
- Enfin, en utilisant le nom d'utilisateur GitHub du développeur, nous croisons les données extraites par les deux processus comme illustré sur la Figure II.7.3. Les deux jeux de données sont mis en correspondance à l'aide des noms d'utilisateurs GitHub présents dans chacun des jeux de données. Ce croisement nous permet d'effectuer une analyse afin de répondre à la Question de recherche 7.1.

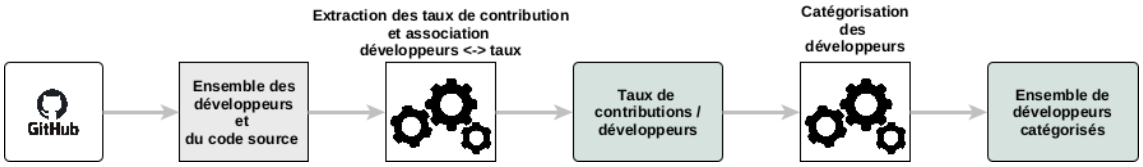



FIGURE II.7.2 – Processus de catégorisation des développeurs, sur la base des travaux du Chapitre II.5.

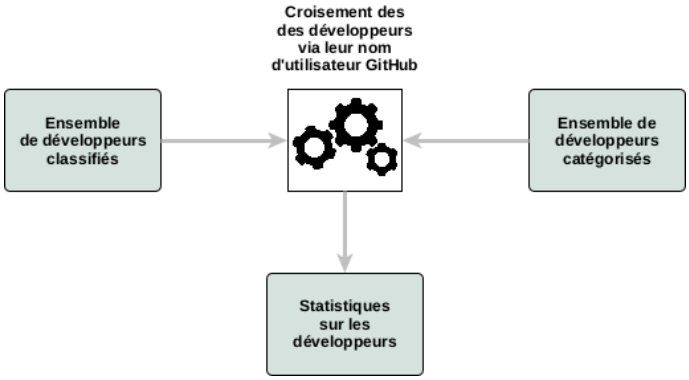



FIGURE II.7.3 – Processus de croisement des données utilisé pour chaque version du projet étudié.

Pour répondre à la Question de recherche 7.2, notre analyse se focalise sur les caractéristiques (variables) qui font que les développeurs passent de la catégorie des développeurs non-expérimentés (NSSE) à celle des développeurs expérimentés (SSE) au cours du temps. Pour cela, et comme montré par la Figure II.7.4, nous observons et analysons spécifiquement les développeurs qui basculent d'une catégorie à l'autre entre une version $V_n$ et la version suivante $V_{n+1}$. Puis, nous observons l'évolution des caractéristiques du développeur entre ces deux versions.

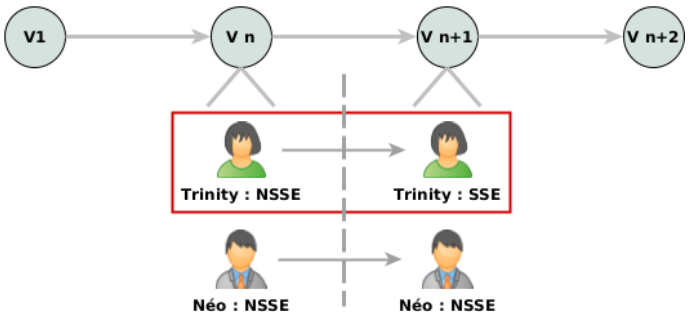



FIGURE II.7.4 – Illustration du changement de catégorie d'une développeuse entre la version $V_n$ et la version $V_{n+1}$.

## 7.3 Résultats

Dans la Section 7.2, nous avons présenté la méthodologie que nous avons appliquée afin de répondre aux questions de recherche 7.1 et 7.2. Dans cette section nous présentons les différents résultats obtenus.

### 7.3.1 Distribution des développeurs expérimentés par rapport à leurs contributions architecturales

Pour répondre à la Question de recherche 7.1, nous avons croisé les résultats des processus illustrés par les Figures II.7.1 et II.7.2 afin d'analyser la répartition des développeurs expérimentés en termes de contributions architecturales.
La Figure II.7.5 présente l'évolution du nombre de développeurs non-expérimentés, de développeurs expérimentés et de lignes de code au fil des versions du projet *BroadleafCommerce*. De manière assez surprenante le nombre de développeurs expérimentés est supérieur à celui des développeurs non-expérimentés entre les versions 2.0.0 et 3.0.0. Cette différence peut s'expliquer par le peu de participation de développeurs "extérieurs" ne faisant pas partie du noyau de développeurs expérimentés appartenant, pour la plupart, à l'entreprise *BroadleafCommerce*. À partir de la version 3.1.0, on observe un croisement des courbes SSE et NSSE qui suit l'augmentation de la taille du projet. L'augmentation du nombre de développeurs dans les deux catégories suit la même tendance que celle du nombre de lignes de code, ce qui peut être un signe de bonne gestion du projet. Nous avons mesuré la corrélation entre le nombre de développeurs expérimentés et la taille du projet à l'aide du coefficient de corrélation de Spearman au risque $\alpha = 0.05$. La valeur de corrélation obtenue est d'environ 0.642 pour une valeur de $p \simeq 0.017$. Comme $p < \alpha$, la valeur de corrélation obtenue est significative. La corrélation (0.642) est, quant à elle, modérée mais non négligeable. Cela prouve l'existence d'un lien entre la taille du projet et le nombre de développeurs expérimentés travaillant sur le projet.

La Figure II.7.6 présente les catégories de contributeurs à l'architecture qui composent les développeurs expérimentés identifiés à l'aide du classifieur sur le projet *BroadleafCommerce*. La Figure II.7.6 montre que, parmi les développeurs expérimentés identifiés, une part fixe des développeurs émarge dans les catégories des *contributeurs architecturaux majeurs* et *contributeurs majeurs* (`CMAR_CM`), ce qui est cohérent avec les constatations faites dans le Chapitre II.5. Une deuxième catégorie de développeurs identifiés parmi les développeurs expérimentés concerne les *contributeurs architecturaux majeurs* et *contributeurs mineurs* (`CMAR_Cm`). Il semble, en effet, normal de détecter cette catégorie au sein des développeurs expérimentés, ces derniers ayant potentiellement une forte contribution à l'architecture de par leurs connaissances. De manière surprenante, la part des *contributeurs architecturaux majeurs* et *contributeurs mineurs* (`CMAR_Cm`) n'est pas constante dans le temps. En effet, à partir de la version 3.1.0, cette catégorie de développeurs cesse d'apparaître dans les développeurs expérimentés ce qui peut s'expliquer par une migration des développeurs dans

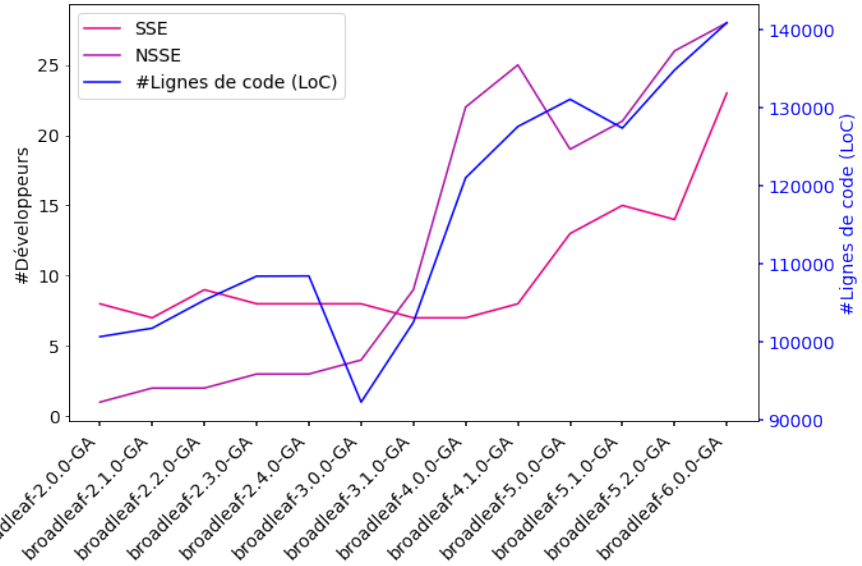



FIGURE II.7.5 – Évolution du nombre de développeurs non-expérimentés (NSSE), expérimentés (SSE) et du nombre de lignes de code (LoC) au cours des version de *BroadleafCommerce*.

d'autres catégories. Un autre aspect que nous pouvons observer concerne l'augmentation progressive du nombre de *contributeurs architecturaux mineurs* et *contributeurs mineurs* (`CmAR_Cm`). Ce nombre est stable de la version 2.0.0 à la version 2.4.0 puis double ensuite à chaque changement de version majeure pour représenter quasiment la moitié des développeurs expérimentés pour chaque version. La présence de cette catégorie peut s'expliquer par le fait que certains développeurs ont des compétences dans d'autres domaines (Maven/Gradle ou structure Java) et sont donc identifiés par le classifieur comme expérimentés. De plus, bien que fiable en termes de mesure F1, il n'est pas exclu qu'il y ait des erreurs de classification. Il existe également des développeurs que nous ne sommes pas parvenus à croiser dans les approches (catégorie inconnue) en raison de différences entre les noms des développeurs. Néanmoins, au regard des constatations que nous faisons ici, notre méthode de classification automatique des développeurs expérimentés est pertinente.

### 7.3.2 Évolution des caractéristiques des développeurs qui gagnent en expérience

Pour répondre à la Question de recherche 7.2 nous avons analysé l'évolution des caractéristiques (variables) lors des changements de catégorie des développeurs. Les Figures II.7.7 et II.7.8 présentent l'évolution des 23 variables (décrites au Chapitre II.6) de 22 développeurs passés de la catégorie des développeurs non-expérimentés (NSSE) à la catégorie des développeurs expérimentés (SSE) dans l'historique des 13 versions de *BroadleafCommerce*. À des fins de visualisation, nous utilisons une échelle logarithmique pour cinq des variables (DiP, ICT, NoC, AddLOC, DelLOC). Les Figures II.7.7 et II.7.8 présentent les évolutions sous forme de boîtes de Tukey. Ces boîtes

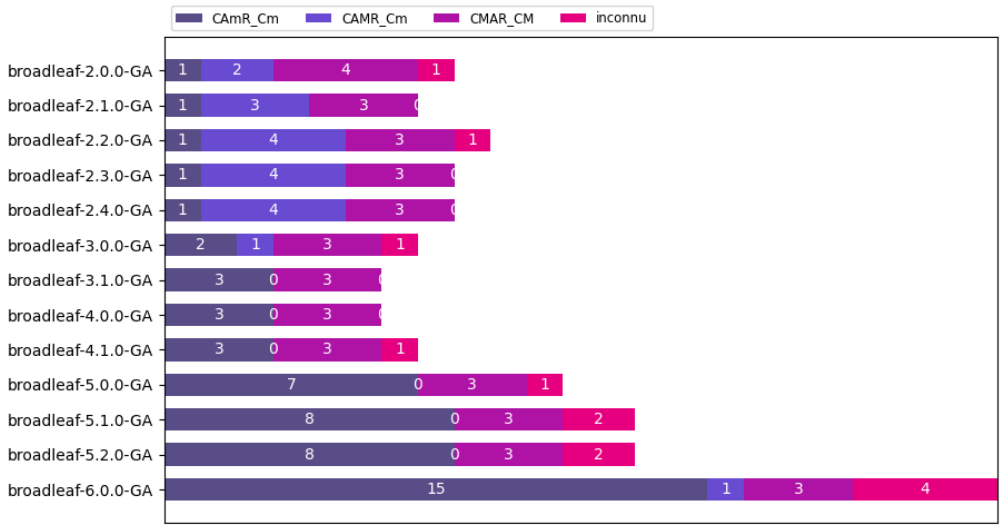



FIGURE II.7.6 – Développeurs expérimentés par catégories de contributeurs.

permettent de comparer les évolutions des variables entre des développeurs avant leur transition (NSSE) puis après leur transition (SSE). Prenons l'exemple de la variable ChurnSAM de la Figure II.7.8. La ligne centrale de la boîte représente la médiane (ici 30). Le premier et troisième quartile (respectivement 7 et 63) se trouvent sur les bords de la boîte. Enfin, chaque extrémité représente une valeur extrême minimale (ici 0) ou maximale (ici 98).

Une première constatation sur le plan architectural est la forte évolution des variables liées à l'architecture *runtime* Spring à savoir : le nombre de lignes spécifiques à Spring ajoutées (AddSAM) ou supprimées (DelSAM) et du *churn* (churnSAM). Le corollaire de cet engagement dans l'architecture Spring est la forte progression du nombre total de lignes de code ajoutées (AddLOC) ou supprimées (DelLOC) et du *churn* (churnLOC). Une autre progression marquée concerne les variables de la structure Java du projet :

- nombre de classes abstraites créées (NoAB),
- nombre de classes non-abstraites créées (NonAB),
- nombre de classes implémentant une interface créées (NoCII),
- nombre de classes créées n'implémentant pas une interface (NoCnII),
- nombre de classes créées étendant une autre classe (NoCE),
- nombre de classes créées n'étendant pas une autre classe (NonCE),
- nombre d'interfaces créées étendant une autre interface (NoIEI),
- nombre d'interfaces créées n'étendant pas une autre interface (NoInEI).

L'ensemble des ces augmentations montre l'aspect multi-factoriel de la transition du développeur de la catégorie des développeurs non-expérimentés à celle des développeurs expérimentés. Les développeurs expérimentés progressent à la fois en termes d'implication dans l'architecture Spring, dans la structure Java du projet et ont une implication globale sur le code nettement plus importante. Un autre aspect développé par les développeurs expérimentés porte sur la compétence Maven/Gradle (AddLGM, DelLGM). En effet, on note là aussi une augmentation montrant que les développeurs gagnent en compétence sur les aspects de gestion du projet, de ses dépendances et de sa construction.

## 7.4 Incertitudes sur la validité

Les résultats que nous obtenons ici doivent être mis en perspective avec les incertitudes sur leur validité. Une partie de ces incertitudes a été décrite dans la Section 5.6.2.

#### Validité externe

Concernant la validité externe, nos résultats se limitent à une preuve de concept. De fait, nous obtenons les mêmes incertitudes sur la validité externe que celles décrites en Section 5.6.2 à savoir, la nécessité d'une diversification de l'étude sur d'autres projets, d'autres technologies d'architectures ou d'autres langages.

#### Validité interne

À des fins d'économie de temps de calcul, nous extrayons de nouvelles données sur les mêmes versions (*General Availability*) que celles du Chapitre II.5. Nous avons donc un échantillonnage assez lâche de l'évolution des développeurs. Un biais d'échantillonnage est donc possible.

#### Incertitude statistique

L'incertitude statistique concerne la confiance et la pertinence accordées aux résultats. Notre classifieur de développeurs n'ayant pas une mesure F1 de 1, il peut subsister des erreurs de classification. De plus, nous utilisons un coefficient de corrélation de Spearman avec un risque de 5% d'obtenir un résultat identique avec des valeurs aléatoires. Cependant, ce niveau de risque est communément admis dans d'autres études empiriques.

## 7.5 Synthèse, limites et perspectives

Dans ce chapitre, nous avons présenté une étude qualitative des caractéristiques des développeurs expérimentés. Nous avons observé l'évolution de 23 variables (métriques) de 22 développeurs passant du statut de développeur non-expérimenté à celui de développeur expérimenté. Pour cela, nous avons mis en place une méthodologie permettant de croiser les données issues de deux de nos

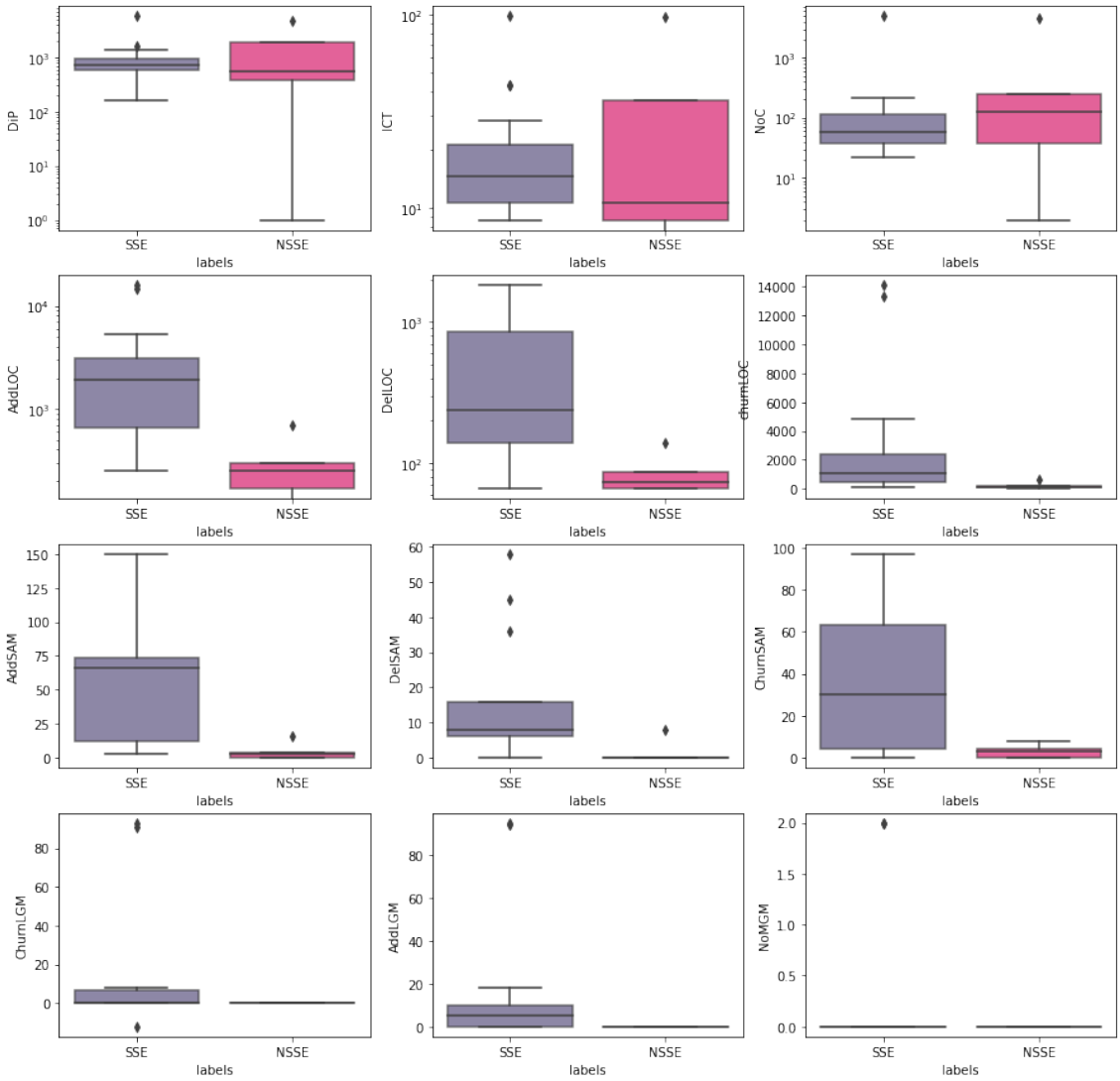



FIGURE II.7.7 – Évolution des variables de 22 développeurs après transition entre la catégorie des non-expérimentés (SSE) et expérimentés (SSE).

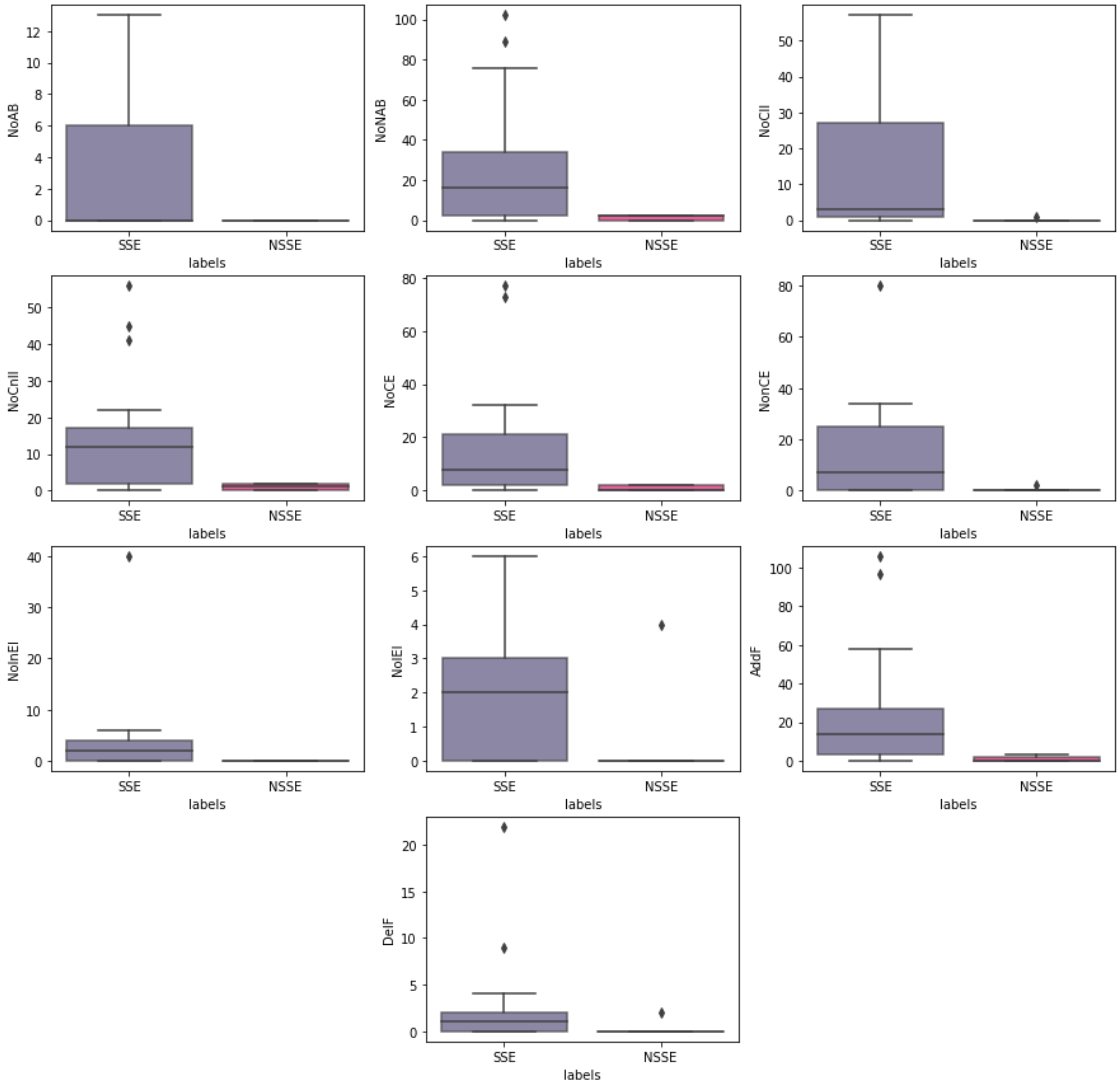



FIGURE II.7.8 – Évolution des variables de 22 développeurs après transition entre la catégorie des non-expérimentés (SSE) et expérimentés (SSE).

précédentes propositions. Les travaux présentés ici et les conclusions que nous en tirons ouvrent plusieurs perspectives.

### 7.5.1 Synthèse

**Synthèse Question de recherche 7.1**

Afin de répondre à la Question de recherche 7.1, nous avons croisé les données issues des travaux des Chapitres II.5 et II.6. Les résultats obtenus montrent l'existence d'une corrélation entre la taille du code, mesurée en nombre de lignes, et le nombre de développeurs expérimentés qui ont été catégorisés par le classifieur créé au Chapitre II.6. Nous montrons également l'existence, parmi les développeurs expérimentés, de développeurs contribuant fortement au code et à l'architecture et ce, tout au long des versions du projet *BroadleafCommerce*. Une autre catégorie composant les développeurs expérimentés concerne les développeurs contribuant soit au code, soit à l'architecture de manière importante. Ce regard croisé entre nos résultats fournit un indice de fiabilité des deux propositions précédentes : métriques de contribution et classifieur de développeurs.

**Synthèse Question de recherche 7.2**

Pour répondre à la Question de recherche 7.2, nous avons étudié l'évolution des caractéristiques des développeurs lors de leur passage du statut de développeur non-expérimenté à celui de développeur expérimenté. Comme nous l'avons vu dans le contexte (Chapitre I.2), cette évolution se manifeste de façon plurielle du point de vue des métriques. En effet, nous notons une augmentation de l'ensemble des métriques des développeurs lors du changement de catégorie. Cette évolution est marquée sur les métriques liées à l'architecture *runtime* montrant ainsi la préoccupation architecturale dont font preuve les développeurs expérimentés. Une autre grande évolution concerne la structure Java du projet. De manière intuitive, nous pouvons déduire de ces évolutions que les développeurs ont une connaissance plus grande et plus vaste du projet de sorte qu'ils sont capables de travailler à la fois sur l'architecture *runtime* et la structure du code Java.

### 7.5.2 Incertitudes sur la validité

#### 7.5.2.1 Validité externe

Nous avons fait une étude croisée de résultats obtenus par deux approches différentes. Cette étude utilise des résultats issus d'un seul projet *open-source*, *BroadleafCommerce*, comme objet d'analyse. Pour confirmer nos résultats et disposer d'une proposition généralisable, il serait nécessaire d'étudier un plus grand nombre de projets. De plus, nous focalisons notre étude croisée sur un type

de projet particulier qui utilise le langage Java ainsi que le *framework* Spring. Afin de généraliser les résultats il serait donc également nécessaire d'étendre l'étude à d'autres types de projets (écrits en C++ ou Javascript, par exemple) et/ou à différents types de *frameworks* (Apache Struts, JEE, OSGI, etc.).

#### 7.5.2.2 Validité interne

Dans un soucis d'économie de temps de calcul, nous avons volontairement échantillonné les versions de *BroadleafCommerce* en choisissant celles étiquetées *General Availability*. Cela pourrait constituer un biais, bien que nos mesures aient été faites sur l'historique complet des changements.

### 7.5.3 Limites

La principale limitation de ce chapitre est la taille de l'évaluation que nous conduisons. En effet, nous n'étudions qu'un seul projet (*BroadleafCommerce*) sur un panel de 13 versions. De plus, nous étudions qu'une seule technologie, le *framework* Spring dans un seul langage (Java).

### 7.5.4 Perspectives

Dans le cadre de ces travaux, une première perspective concerne l'élargissement de l'étude à un panel de projets de même nature que celui expertisé ici, à savoir, des projets utilisant le langage Java et le *framework* Spring. Une extension à d'autres langages de programmation et à d'autres *frameworks* d'architectures est également envisageable. Cela permettrait de conforter les résultats obtenus ici, mais aussi d'étudier les politiques de gestion des développeurs dans les projets en fonction des technologies.
Une deuxième perspective concerne la prédiction du nombre de développeurs expérimentés en fonction de la taille du projet et ou de ses technologies architecturales. Nous avons vu dans ce chapitre qu'il y a une corrélation entre la taille du projet et le nombre de développeurs expérimentés. Avec plus de données, il serait intéressant d'explorer ce lien pour la création d'un prédicteur du nombre de développeurs expérimentés se basant sur la taille du projet.
Enfin, une troisième perspective concerne la prédiction de la capacité d'un développeur non-expérimenté à devenir expérimenté. Connaissant les évolutions des caractéristiques des développeurs entre les deux catégories, il serait possible de prédire le potentiel du développeurs non-expérimenté à devenir expérimenté. Tout comme la perspective précédente, cela nécessite un volume de données important dont nous ne disposons pas ici.

# Chapitre 8

# Implémentations et reproductibilité

**Sommaire**

Dans ce chapitre, nous donnons les éléments permettant l'accès aux données et aux implémentations que nous avons utilisées et produites dans cette thèse. Nous effectuons également, pour chacune de nos contributions, une évaluation de sa reproductiblité selon les cadres méthodologiques de González-Barahona et Robles [GR12] pour le génie logiciel empiriques et Olorisade *et al.* [OBA17] pour l'apprentissage machine.

## 8.1 Intégrité scientifique et reproductibilité

L'éthique, la déontologie et l'intégrité scientifique sont les piliers sur lesquels le chercheur doit appuyer ses recherches. Ces composantes indissociables permettent au chercheur d'inscrire ses recherches et dans un cadre moral, éthique et intègre qui est à la fois réglementaire et acceptable par la société. La charte française de déontologie des métiers de la recherche [1] éditée en 2015 par le CNRS et signée par de nombreux acteurs de la recherche, présente les principes clefs de ces trois aspects. La reproductibilité est une composante de l'intégrité scientifique. Dans sa charte, le CNRS la définit comme suit :

> *"La description du protocole expérimental, dans le cadre de cahiers de laboratoires par exemple, doit permettre la reproductibilité des faits expérimentaux."*

Cette définition montre que les éléments permettant la reproduction des expériences doivent être suffisamment décrits et/ou mis à disposition.

### 8.1.1 Reproductibilité en génie logiciel empirique

Les problématiques de la reproductibilité en informatique, et plus précisément dans le domaine du génie logiciel empirique, ont été mises en avant en 2012 par González-Barahona et Robles [GR12]. Les auteurs ont créé un cadre méthodologique permettant d'évaluer la reproductibilité des études empiriques. Ils ont, pour ce faire, identifiés huit éléments manipulés dans des études de génie logiciel empirique et présentés sur la Figure 8.1 :

1. Source de données (*data source*). Cette source peut-être le ou les dépôt(s) logiciels utilisé(s) pour l'étude. Cette source peut concerner également des objets d'études (bibliothèques, *packages*, etc.) ou un ensemble de données concaténées.
2. Méthodologie de récupération des données (*retrieval methodology*). Opération d'extraction depuis la source de données des données brutes, à l'aide d'outils manuels ou automatisés.
3. Jeu de données brutes (*raw dataset*). Données obtenues après extraction sans aucune modification.
4. Méthodologie d'extraction (*methodology extraction*). Processus mis en œuvre pour extraire et nettoyer les données pertinentes des données brutes.

1. https://comite-ethique.cnrs.fr/wp-content/uploads/2020/01/2015_Charte_nationale_d%C3%A9ontologie_190613.pdf

5. Paramètres de l'étude (*study parameters*). Les paramètres de l'étude influant sur les processus modifiant le jeu de données. Il peut s'agir de périodes de temps, de types d'informations, etc.
6. Ensemble de données traitées (*processed dataset*). Ensemble de données traitées via la méthodologie d'extraction qui seront les entrées de l'analyse méthodologique. Ces données peuvent être, par exemple, une base de données SQL, un fichier CSV utilisable dans un tableur ou dans tout autre outil d'analyse.
7. Méthodologie d'analyse (*analysis methodology*). Processus d'analyse de tout ou partie des données en vue d'obtenir des résultats.
8. Jeu de données de résultats (*results dataset*). Données produites par la méthodologie d'analyse et en utilisant l'ensemble des données traitées. Ces données sont la base des résultats de recherche. Elles peuvent prendre différentes formes : textuelle, graphique, mathématique, etc.

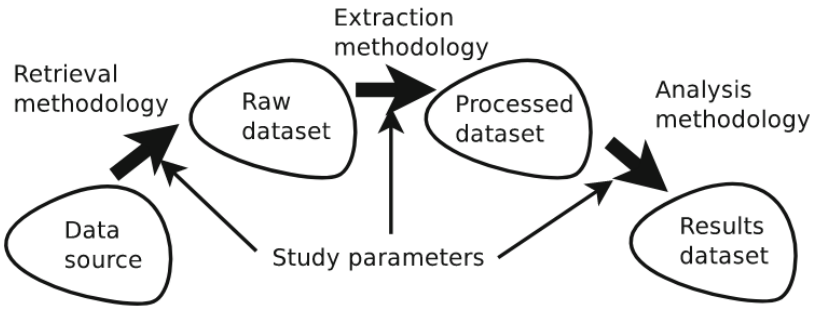



FIGURE 8.1 – Éléments ayant un impact sur la reproductibilité organisés selon le sens de leurs relations durant le processus de recherche [GR12].

González-Barahona et Robles ont décrit cinq attributs applicables sur les éléments de recherche et ayant un effet sur la reproductibilité des études :

- L'identification des éléments : endroit où sont stockés les éléments de l'étude, c'est-à-dire où leurs originaux peuvent être trouvés.
- La description des travaux : le niveau de détail des informations publiées à propos des éléments de recherche.
- La disponibilité : facilité avec laquelle d'autres chercheurs peuvent obtenir les éléments de recherche.
- La persistance : pérennité des éléments de recherche dans le temps.
- La flexibilité : adaptabilité des éléments de recherche à de nouveaux environnements.

Puis, comme le montre la Table 8.1, González-Barahona et Robles [GR12] ont fixé les éléments sur lesquels pouvaient s'appliquer les attributs.
Chacun de ces attributs peut ensuite être évalué via une échelle de valeurs symboliques, comme le montre la Table 8.2 :

| | Source de données | Jeux de données | Paramètres | Outils |
|---|---|---|---|---|
| Identification | ✔ | ✔ | ✔ | ✔ |
| Description | ✔ | ✔ | ✔ | ✔ |
| Disponibilité | ✔ | ✔ | | ✔ |
| Persistance | ✔ | ✔ | | ✔ |
| Fiabilité | | ✔ | | ✔ |

TABLE 8.1 – Application des attributs sur les différents éléments de recherche en génie logiciel empirique, traduit de González-Barahona et Robles [GR12].

| Attribut | Valeurs |
|---|---|
| Identification | Complète, Partielle, Non, N/A |
| Description | Textuelle, code source, Partielle, Non, N/A |
| Disponibilité | Privée, Publique, Non, N/A |
| Persistance | Probable, Inconnue, N/A |
| Flexibilité | Complète, Partielle, Non, N/A |

TABLE 8.2 – Valeurs possibles pour chacun des attributs.

- Complète : valeur utilisée pour décrire la mise à disposition de toutes les informations nécessaires pour localiser ou identifier un élément de l'étude. Les informations données peuvent parfois êtres seulement textuelles ou issues du code source de l'expérimentation.
- Partielle : indique un manque d'informations ou une trop grande généralité autour de l'élément de l'étude.
- Non : absence totale d'information sur l'élément.
- N/A (Non-applicable) : implique que l'attribut n'est pas applicable à l'élément en question.
- Privée/Publique : valeur permettant de caractériser la disponibilité de l'élément de recherche.
- Probable : valeur s'appliquant à l'attribut de persistance dans le cas où il existe une possibilité qu'un élément soit disponible pour un accès futur.
- Inconnu : valeur utilisée pour l'attribut de persistance dans le cas où il est impossible de déterminer une quelconque persistance dans le temps.

En utilisant les éléments et les caractéristiques décrits par [GR12], nous allons évaluer la reproductibilité de chacune de nos contributions dans la suite de ce chapitre.

### 8.1.2 Reproductibilité en apprentissage automatique

L'apprentissage automatique est également une discipline où la notion de reproductibilité tient une place importante. De part sa nature, l'apprentissage automatique se sert de jeux de données, étiquetées ou non, et effectue un ou plusieurs traitement sur ces données. Le résultat d'un processus d'apprentissage automatique est souvent un algorithme entraîné ou ajusté sur les données précédemment traitées. De plus, certaines des étapes du processus sont parfois sujettes à des paramètres aléatoires qu'il convient de maîtriser en vue de les rendre reproductibles. Ainsi, on retrouve

| | Source de données | Jeux de données | Paramètres | Méthodes employées | Outils/Algorithmes |
|---|---|---|---|---|---|
| Identification | ✔ | ✔ | ✔ | ✔ | ✔ |
| Description | ✔ | ✔ | ✔ | ✔ | ✔ |
| Disponibilité | ✔ | ✔ | | | ✔ |
| Persistance | ✔ | ✔ | | | ✔ |
| Flexibilité | | ✔ | | | ✔ |

TABLE 8.3 – Application des attributs sur les différents éléments de recherche en apprentissage automatique [OBA17].

plusieurs similarités entre la recherche empirique en génie logiciel et l'apprentissage automatique : sources de données, traitements statistiques, manipulation de données brutes, etc. C'est sur la base de ces similarités que Olorisade *et al.* [OBA17] ont repris la méthodologie de González-Barahona et Robles [GR12] afin de l'appliquer à l'apprentissage automatique. Olorisade *et al.* ont réutilisé les attributs (Table 8.1), les valeurs que peuvent prendre ces attributs (Table 8.2) et ont modifié les éléments de recherche afin de les adapter au contexte de l'apprentissage automatique.
Les éléments de recherche qu'Olorisade *et al.* ont défini sont :

1. La source de données : identique à la description donnée par González-Barahona et Robles.
2. Le jeu de données brutes : identique à la description donnée par González-Barahona et Robles.
3. Le pré-traitement : manipulation, transformation et/ou suppression de données problématiques. En traitement automatique du langage naturel, l'étape de pré-traitement désigne, par exemple, l'étape de *tokenisation*. Sur des données numériques, un pré-traitement classique concerne la mise à l'échelle des données.
4. La sélection des variables les plus représentatives : approche utilisée pour sélectionner les variables les plus représentatives dans un jeu de données.
5. La réduction de dimension : méthodes utilisées pour réduire la dimensionnalité des données.
6. Le partitionnement des données : méthode de répartition des données utilisées pour l'entraînement et l'évaluation de l'algorithme d'apprentissage automatique.
7. La méthode d'apprentissage automatique : détails du ou des algorithme(s) utilisé(s), du contrôle de l'aléatoire par la fixation de graines mais aussi des différents paramètres et hyperparamètres utilisés.
8. La méthode de validation : approche utilisée pour tester la performance du classifieur.
9. Les *frameworks* et bibliothèques tiers : listes de bibliothèques et *frameworks* utilisés dans le cadre de l'expérimentation.
10. Méthodes personnalisées : détail des méthodes spécifiques proposées par les auteurs de l'expérimentation.

Comme montré par la Table 8.3, à l'image des travaux de González-Barahona et Robles [GR12], Olorisade *et al.* [OBA17] ont défini les éléments de recherche sur lesquels appliquer les cinq attributs.

Avec l'ensemble de ces éléments et attributs, Olorisade *et al.* ont été en mesure d'évaluer la reproductibilité de 33 études liées au traitement du langage naturel.

### 8.1.3 Outils logiciels pour la reproductibilité

La reproductibilité des études en génie logiciel peut être mise en œuvre à l'aide de différents outils. Nous allons présenter ici les outils que nous utilisons.

#### 8.1.3.1 Mise à disposition du code et des données via GitHub

Nous utilisons GitHub en tant que plate-forme d'hébergement de nos différentes données de recherche et du code d'expérimentation. Ce choix est motivé par l'accessibilité de GitHub au public et sa gratuité. GitHub ne donnant pas de garanties à long terme sur la pérennité des données nous avons fait indexer nos données et codes sources par *Software Heritage* [CZ17].

#### 8.1.3.2 Software Heritage

*Software Heritage* [CZ17] est une plate-forme visant à indexer, préserver et conserver le code source produit par l'humanité à la manière de la bibliothèque d'Alexandrie. Le projet a été initié par Roberto Di Cosmo et Stefano Zacchiroli [CZ17] en 2017 avec le soutiens de l'Institut National de Recherche en Informatique et en Automatique (INRIA). *Software Heritage* est conçu de manière à pouvoir pérenniser les dépôts logiciels indexés ainsi que les différents liens pour y accéder. Nous avons fait sauvegarder nos dépôts par la plate-forme et donnons un lien permanent vers *Software Heritage* pour chacun des dépôts GitHub créés.

#### 8.1.3.3 Carnets de code Jupyter

Jupyter est une application web créée en 2014 fournissant une interface web et un interpréteur Python. Jupyter permet de créer des "carnets de code" exécutables via un navigateur web. L'environnement Jupyter permet l'exécution, la visualisation des résultats et leur sauvegarde. Un carnet peut ainsi être exécuté et sauvegardé avec ses résultats (graphiques ou non) sans avoir à ré-exécuter l'application pour les obtenir. Ces différents aspects font de Jupyter un outil intéressant pour la reproductibilité logicielle, en complément de la mise à disposition des données via des plates-formes libre d'accès, et ce d'autant plus que l'étude de Pimentel *et al.* a montré à large échelle le caractère reproductible des carnets [PMBF19].

## 8.2 Implémentations réalisées et données utilisées

La reproductibilité des études fait partie de l'intégrité scientifique du chercheur. Dans cette section, nous allons donner les éléments, codes sources et données, permettant la reproductibilité des expériences menées dans cette thèse.

### 8.2.1 Catégorisation des développeurs à l'aide d'une métrique d'évaluation de la contribution à l'architecture

Dans le Chapitre II.5, nous avons construit et utilisé un modèle de contribution à l'architecture. L'ensemble des données concernant le projet *BroadleafCommerce* ont été extraites par l'API HTTP de GitHub: `https://api.github.com/repos/BroadleafCommerce/BroadleafCommerce`. Nous avons mis les données à disposition de la communauté sur un dépôt GitHub : `https://github.com/DedalArmy/empirical-study-architectural-contributions/data`. Le code permettant de reproduire l'expérimentation et les résultats est disponible sur GitHub : `https://github.com/DedalArmy/empirical-study-architectural-contributions` ou sur *Software Heritage* : `https://archive.softwareheritage.org/browse/origin/directory/?origin_url=https://github.com/DedalArmy/empirical-study-architectural-contributions`.

Nous avons conduit l'évaluation de la reproductibilité de notre étude empirique selon le cadre méthodologique de González-Barahona et Robles [GR12]. Cette évaluation est donnée par la Table 8.4. Nous atteignons une bonne reproductibilité pour cette contribution du fait de la mise à disposition des données brutes utilisées mais également de la mise à disposition du code source lié aux expérimentations. Une piste d'amélioration pourrait être de fournir les différents jeux de données intermédiaires entre les données brutes et les résultats.

| | Identification | Description | Disponibilité | Persistence | Flexibilité |
|---|---|---|---|---|---|
| Source de données | Complète | Textuelle | Publique | Probable | N/A |
| Méthodologie de récupération des données | Partielle | Textuelle | Non | N/A | N/A |
| Jeu de données brutes | Compléte (lien) | Code source + Textuelle | Publique | Probable | Partielle |
| | `https://github.com/DedalArmy/empirical-study-architectural-contributions/data` | | | | |
| Méthodologie d'extraction | Complète | Code source + Textuelle | Publique | Probable | Non |
| | `https://github.com/DedalArmy/empirical-study-architectural-contributions` | | | | |
| Paramètres de l'étude | Partielle | Code source + Textuelle | N/A | N/A | N/A |
| Ensemble de données traitées | Partielle | Textuelle | Non | Inconnue | Partielle |
| Méthodologie d'analyse | Compléte | Code source + Textuelle | Publique | N/A | N/A |
| | `https://github.com/DedalArmy/empirical-study-architectural-contributions` | | | | |
| Jeu de données de résultats | Partielle | Textuelle | Non | Non | Non |

TABLE 8.4 – Évaluation de la reproductibilité de l'étude empirique sur la contribution architecturale des développeurs selon le cadre méthodologique de González-Barahona et Robles [GR12].

### 8.2.2 Classification automatique de tickets de bogues

Dans le Chapitre II.4, nous avons réutilisé le jeu de données mis à disposition par Herzig *et al.* [HJZ13]. Herzig *et al.* n'ont donné que les identifiants et les étiquettes des tickets de bogues https://www.st.cs.uni-saarland.de/softevo/bugclassify/. Nous avons souhaité mettre à disposition la totalité de notre jeu de données comprenant les étiquettes, les identifiants et les tickets (titres et descriptions) afin de faciliter la réutilisation du jeu de données mais également la reproductibilité. Ce jeu de données est disponible à l'adresse suivante : https://github.com/qperez/ATTIC/tree/main/data. L'ensemble du code réalisé dans le cadre de nos travaux (l'algorithme génétique et code lié à l'apprentissage automatique) peut être consulté à l'adresse suivante sur GitHub : https://github.com/qperez/ATTIC ou sur *Software Heritage* : https://archive.softwareheritage.org/browse/origin/directory/?origin_url=https://github.com/qperez/ATTIC. Afin de faciliter l'utilisation du code et la reproduction des expériences, le code d'expérimentation a été rédigé dans des carnets Jupyter.
La Table 8.5 donne l'évaluation de la reproductibilité de cette contribution. Nous obtenons une reproductibilité quasi parfaite. En effet, nous sommes en mesure de fournir l'ensemble des sources permettant la constitution du jeu de données brutes créés par Herzig *et al.* [HJZ13]. Nous avons mis à disposition l'ensemble de notre code, la liste des librairies tierces nécessaires et décrit l'ensemble des méthodes utilisées dans le cadre de ces expérimentations.

#### 8.2.2.1 Prototype d'assistance à la rédaction de tickets

À des fins de pérennité de l'implémentation du prototype d'assistance à la rédaction de tickets (présenté en Section 4.7), un conteneur Docker l'hébergeant est disponible grâce à l'URL suivante : public.ecr.aws/v3b0r6p4/buticc-web-app:latest. Le code du prototype est disponible sur GitHub: https://github.com/qperez/ATTIC-webapp ou sur *Software Heritage* : https://archive.softwareheritage.org/browse/origin/directory/?origin_url=https://github.com/qperez/ATTIC-webapp.

| | Identification | Description | Disponibilité | Persistance | Flexibilité |
|---|---|---|---|---|---|
| Source de données | Complète | Textuelle | Publique | Non | N/A |
| | https://www.st.cs.uni-saarland.de/softevo/bugclassify/ | | | | |
| Jeu de données brutes | Complète | Textuelle | Publique | Probable | Complète |
| | https://github.com/qperez/ATTIC/tree/main/data | | | | |
| Pré-traitement | Complète | Textuelle + Code source | N/A | N/A | N/A |
| Sélection des variables représentatives | Complète | Textuelle + Code source | N/A | N/A | N/A |
| Réduction de dimension | Complète | Textuelle + Code source | N/A | N/A | N/A |
| Partitionnement des données | Complète | Textuelle + Code source | N/A | N/A | N/A |
| Méthode d'apprentissage | Complète | Textuelle + Code source | N/A | N/A | N/A |
| Méthode de validation | Complète | Textuelle + Code source | N/A | N/A | N/A |
| *Frameworks* et bibliothèques | Complète | Textuelle + Code source | Publique | Probable | Complète |
| | https://github.com/qperez/ATTIC/blob/main/requirements.txt | | | | |
| Méthodes personnalisées (algorithme génétique) | Complète | Textuelle + Code source | Publique | Probable | Complète |
| | https://github.com/qperez/ATTIC/tree/main/genetic_algorithm | | | | |

TABLE 8.5 – Évaluation de la reproductibilité de l'étude concernant la classification de tickets selon le cadre méthodologique de Olorisade *et al.* [OBA17].

### 8.2.3 Classification des développeurs expérimentés

Dans le Chapitre II.6, nous avons créé un jeu de données de développeurs expérimentés ou non depuis un ensemble de projets GitHub. Nous avons ensuite utilisé ce jeu de données afin d'entraîner un classifieur. Nous mettons à disposition l'ensemble des éléments ayant servi à l'expérimentation. Le jeu de données anonymisées que nous avons créé est accessible à l'adresse suivante : https://anonymous.4open.science/r/EDF-CCF1/dataset_dev_anonymized.csv. L'ensemble du code d'expérimentation est récupérable ici : https://anonymous.4open.science/r/EDF-CCF1/. À des fins de reproductibilité, l'ensemble de ce code a été réalisé dans des carnets Jupyter.

La Table 8.6 donne l'évaluation de la reproductibilité de cette contribution. Nous obtenons une bonne reproductibilité de nos travaux notamment sur l'ensemble des méthodes et outils utilisés. L'anonymisation de nos données nous contraint à masquer les sources permettant d'identifier de manière personnelle un développeur. Cela a pour effet une moindre identification des sources de données ayant été utilisées pour constituer le jeu.

| | Identification | Description | Disponibilité | Persistence | Flexibilité |
|---|---|---|---|---|---|
| Source de données | Partielle | Textuelle | Publique | Non | N/A |
| Jeu de données brutes | Partielle | Textuelle | Publique (anonymisée) | Probable | Complète |
| | https://anonymous.4open.science/r/EDF-CCF1/dataset_dev_anonymized.csv | | | | |
| Pré-traitement | Complète | Textuelle + Code source | N/A | N/A | N/A |
| Sélection des variables représentatives | Complète | Textuelle + Code source | N/A | N/A | N/A |
| Réduction de dimension | Complète | Textuelle + Code source | N/A | N/A | N/A |
| Partitionnement des données | Complète | Textuelle + Code source | N/A | N/A | N/A |
| Méthode d'apprentissage | Complète | Textuelle + Code source | N/A | N/A | N/A |
| Méthode de validation | Complète | Textuelle + Code source | N/A | N/A | N/A |
| *Frameworks* et bibliothèques | Complète | Textuelle + Code source | Publique | Probable | Complète |
| | https://anonymous.4open.science/r/EDF-CCF1/requirements.txt | | | | |
| Méthodes personnalisées | Non | Non | Non | Non | Non |

TABLE 8.6 – Évaluation de la reproductibilité de l'étude concernant la classification des développeurs selon le cadre méthodologique de Olorisade *et al.* [OBA17].

### 8.2.4 Étude des profils de développeurs expérimentés par croisement de données et apprentissage machine

Cette étude empirique a été réalisée en utilisant les données obtenues aux Chapitres II.5 et II.6. Elle nous a permis de mettre en lumière les différentes catégories de contributeurs à l'architecture parmi les développeurs classés comme expérimentés par notre classifieur. Nous avons également étudié la corrélation entre la taille du code et le nombre de développeurs expérimentés. Enfin, nous avons étudié l'évolution des caractéristiques des développeurs lors de leurs transition de non-expérimenté vers expérimenté. La Table 8.7 donne l'évaluation de la reproductibilité de cette étude empirique. Réutilisant les données de l'étude empirique sur la contribution architecturale des développeurs, nous ne pouvons améliorer la méthodologie de récupération des données. Nous avons cependant travaillé à obtenir une bonne reproductibilité globale en fournissant les carnets Jupyter qui permettent de conduire l'étude depuis les données brutes jusqu'aux résultats.

| | Identification | Description | Disponibilité | Persistence | Flexibilité |
|---|---|---|---|---|---|
| Source de données | Complète | Textuelle | Publique | Probable | N/A |
| Méthodologie de récupération des données | Partielle | Textuelle | Non | N/A | N/A |
| Jeu de données brutes | Complète (lien) | Code source + Textuelle | Publique | Probable | Partielle |
| | https://anonymous.4open.science/r/cross_study_developers-470C/data | | | | |
| Méthodologie d'extraction | Complète | Code source + Textuelle | Publique | Probable | Non |
| | https://anonymous.4open.science/r/cross_study_developers-470C | | | | |
| Paramètres de l'étude | Partielle | Code source + Textuelle | N/A | N/A | N/A |
| Ensemble de données traitées | Partielle | Textuelle | Non | Inconnue | Partielle |
| Méthodologie d'analyse | Compléte | Code source + Textuelle | Publique | N/A | N/A |
| | https://anonymous.4open.science/r/cross_study_developers-470C | | | | |
| Jeu de données de résultats | Compléte | Code source + Textuelle | Publique | Probable | Partielle |

TABLE 8.7 – Évaluation de la reproductibilité de l'étude selon le cadre méthodologique de González-Barahona et Robles [GR12].

# Chapitre 9

# Conclusions et perspectives

**Sommaire**

Cette thèse présente nos propositions pour la gestion des contributions architecturales dans les projets logiciels grâce à la création de métriques, à des analyses empiriques et à la mise en œuvre d'apprentissage machine. Ce chapitre de conclusion résume les contributions de cette thèse et donne des perspectives de recherche.

## 9.1 Contributions de cette thèse

Cette thèse contribue à la recherche en génie logiciel empirique avec une focalisation particulière sur l'architecture logicielle et les contributions des développeurs à cette dernière.

### 9.1.1 Classification automatique des tickets de bogues

Une première contribution concerne la création d'un classifieur de tickets de bogues avec des performances supérieures à l'état de l'art. Cette contribution s'appuie sur des travaux antérieurs [THPM17, PHM13, CS15, PSHS17, QS18, PSHS17] et sur le jeu de 5591 tickets étiquetés manuellement par Herzig *et al.* [HJZ13]. Nous avons créé un classifieur automatique de tickets permettant d'identifier ceux décrivant des bogues. Pour cela, plusieurs méthodes ont été combinées afin d'atteindre de bonnes performances de classification. Le corpus de tickets a d'abord été nettoyé en supprimant les mots sur-représentés et sous-représentés puis il a été converti en vecteurs en utilisant la méthode TF-IDF. Nous avons ensuite sélectionné les variables les plus représentatives à l'aide de la méthode du chi-deux. Nous avons évalué un panel de six classifieurs préalablement optimisés à l'aide d'une méthode combinatoire afin de comparer leurs performances de classification sur le corpus. Nous avons ainsi sélectionné le meilleur candidat qui est un réseau de neurones multi-couches. Enfin, nous avons optimisé ce classifieur à l'aide d'un algorithme génétique. L'ensemble des ces étapes nous permet d'obtenir de très bonnes performances de classification sur le corpus de Herzig *et al.*.
Nous avons également confronté notre classifieur à des tickets inconnus issus de trois projets *open-source* disponibles sur GitHub. Nos résultats montrent de bonnes performances sur les projets utilisant les mêmes technologies que le jeu de tickets d'entraînement donné par Herzig *et al.* (langage Java). En revanche, nous obtenons des performances médiocres sur les tickets d'un projet utilisant des technologies différentes (notamment le langage TypeScript). Cela suggère une forme de sur-ajustement de notre classifieur qui le rend moins performant pour d'autres technologies.
Enfin, en utilisant des méthodes d'explicabilité, nous avons observé les termes influant sur la classification des tickets. L'ensemble de ces résultats suggère que notre classifieur peut être utilisé comme indicateur de qualité, notamment pour le dénombrement des tickets de bogues, dont l'évolution au fil du projet est un indicateur intéressant. Une autre application concerne l'assistance à la rédaction de tickets. Nous avons créé un prototype d'assistant s'appuyant sur notre classifieur. Il permet la rédaction d'un ticket complet, avec un titre et une description, et fournit la prédiction de sa classe (bogue ou non), indiquant ainsi si le ticket est clairement rédigé ou ambigu. Ce prototype a été mis à disposition en ligne.

### 9.1.2 Mesure de contribution à l'architecture

Une deuxième contribution est l'étude d'une métrique d'évaluation de la contribution des développeurs à l'architecture. Cette métrique est basée sur les travaux de Bird *et al.* [BNM+11] mais également ceux de Foucault [FFB14]. La formalisation de la métrique de contribution proposée par Foucault *et al.* nous a permis sa réutilisation et son adaptation à notre problématique, à savoir mesurer la contribution globale d'un développeur à l'architecture *runtime* dans un projet. La métrique initiale de Foucault *et al.* mesurant la contribution à une granularité plus réduite (fichier ou *package*), nous l'avons adaptée pour mesurer la contribution globale sur le projet. Sur la base de cette adaptation, nous avons créé une métrique d'évaluation des contributions à l'architecture *runtime*. Nous avons réutilisé les travaux de Bird *et al.* [BNM+11] pour catégoriser les développeurs en fonction de leur niveau de contribution (déterminé grâce à un seuil de contribution) à la fois au code global mais également à l'architecture. Nous avons conduit une expérimentation sur treize versions d'un projet industriel *open-source*. Cette expérimentation a permis de mettre en lumière différentes catégories de développeurs en fonction de leur taux de contribution à la fois au code mais également à l'architecture. Nous avons également montré par examen du *turn-over* qu'il existe un petit noyau pérenne de développeurs contribuant beaucoup à l'architecture et au code tout au long des versions. D'autres profils de développeurs coexistent : certains sont très impliqués dans le code global mais peu sur l'architecture et inversement. Nous constatons également qu'un grand nombre de développeurs n'apportent que de très petites contributions. De fait, nous avons pu valider la pertinence de la métrique proposée pour l'évaluation de la contribution architecturale et des seuils permettant de catégoriser les développeurs.

### 9.1.3 Classification automatique de développeurs expérimentés

Une troisième contribution concerne la classification automatique des développeurs expérimentés à l'aide d'une méthode d'apprentissage supervisé. Les développeurs expérimentés sont souvent ceux portant la connaissance historique et technique des projets logiciels. Ainsi, leur départ peut être la source d'une réelle perte de connaissances. Repérer les développeurs expérimentés est d'autant plus important que la connaissance technique acquise au long cours revêt bien souvent un caractère architectural et atteste d'une pluralité de compétences [McB07]. La création de ce système de classification de développeurs s'est fait en plusieurs étapes.

Ne disposant pas d'un jeu de données étiquetées de développeurs, nous l'avons créé. Pour cela, nous avons extrait 703 développeurs issus de 17 projets *open-source* utilisant le *framework* d'architecture Spring. Nous avons ensuite, pour chacun de ces 703 développeurs, calculé 23 métriques logicielles. Puis, par des recherches systématiques sur internet, nous avons étiqueté manuellement l'ensemble des développeurs avec les rôles qu'ils occupent dans les projets. Nous avons ensuite classé les développeurs en deux catégories en fonction de leurs rôles : développeur expérimenté ou non-expérimenté.

Ensuite, nous avons nettoyé le jeu de données par élimination des valeurs aberrantes. Un travail

a été effectué sur les 23 variables afin de réduire leur dispersion et normaliser leurs valeurs. Afin de palier à un déséquilibre dans nos données, à savoir peu de développeurs expérimentés comparativement aux développeurs non expérimentés, nous avons utilisé une méthode de génération de données synthétiques pour améliorer l'équilibre du jeu de données (*k-means* SMOTE [LDB17]). Nous avons ensuite comparé les résultats d'un panel de six classifieurs préalablement optimisés à l'aide d'une méthode combinatoire. Suite à ce comparatif, le classifieur *random forest* s'est avéré être le plus performant. Nous obtenons de bons résultats de classification avec une mesure F1 de 0.754, un rappel de 0.7033 et une précision de 0.8411.

Enfin, en utilisant des méthodes d'explicabilité du classifieur, nous avons observé quelles étaient les variables influençant le plus le classifieur. Nous avons montré que, contrairement à notre intuition, les variables concernant l'architecture *runtime* étaient celles ayant une moindre influence sur la décision de classification. En revanche, les variables concernant la structure Java du projet ou encore le nombre de fichiers supprimés ont une influence plus forte. Cela pourrait indiquer que les développeurs expérimentés font beaucoup de modifications sur la structure du projet du fait de leurs connaissances et de leur expérience.

### 9.1.4 Étude des profils des développeurs expérimentés

Enfin, une dernière contribution concerne l'étude des profils des développeurs expérimentés que nous sommes capables de détecter à l'aide de notre classifieur de développeurs. Nous croisons ici les résultats de classification automatique avec ceux obtenus par mesure des contributions globales et architecturales. Ainsi, nous sommes en mesure d'étudier les catégories de développeurs qui composent les développeurs classés comme expérimentés. Nos résultats montrent que parmi les développeurs expérimentés, nous retrouvons un nombre non-négligeable de développeurs contribuant majoritairement au code global de l'application mais aussi à l'architecture. Il existe également une corrélation entre la taille du code et le nombre de développeurs expérimentés. Les résultats mettent en lumière la pertinence de notre classifieur comme détecteur de développeurs expérimentés ayant également des compétences en architecture.

L'étude du changement de catégorie des développeurs (transition de la catégorie non-expérimenté vers la catégorie expérimenté) montre une évolution conjointe des 23 variables utilisées pour caractériser les développeurs. Lors du changement de catégorie, nous notons une évolution importante des métriques liées à l'architecture *runtime*, montrant ainsi la préoccupation architecturale dont font preuve les développeurs expérimentés. Une autre évolution importante concerne les variables relatives à la structure Java du projet. Ainsi, nous pouvons déduire de ces évolutions que les développeurs qui deviennent expérimentés ont une connaissance large et transversale du projet et sont donc capables de travailler à la fois sur l'architecture *runtime* et la structure du code Java.

## 9.2 Limitations et perspectives

L'énoncé des limites des contributions de cette thèse nous permet de présenter des perspectives à ces travaux.

### 9.2.1 Perspectives liées à la classification automatique de tickets

L'approche de classification que nous proposons pour identifier les tickets de bogue présente certaines limitations. Une première limitation provient du fait que le classifieur est sensible aux technologies. En effet, ce classifieur a été entraîné sur un jeu de tickets Java et nous avons montré que, dans un autre contexte technologique, les résultats de classification subissaient une baisse importante.
Une deuxième limitation est liée au nombre de catégories que nous sommes capables d'identifier. Actuellement, nous sommes en mesure d'effectuer exclusivement une classification binaire des tickets (bogues ou non).
Une première perspective concerne l'augmentation du nombre et de la diversité des tickets permettant l'entraînement du classifieur. Cela pourrait permettre son utilisation dans un environnement générique de production de logiciels.
Une seconde perspective serait la spécialisation de notre classifieur sur des tickets décrivant des bogues liées à l'architecture logicielle. Cette spécialisation permettrait l'identification de développeurs ayant des compétences architecturales. Ce classifieur spécialisé pourrait également servir à l'orientation des tickets vers les développeurs ayant des compétences architecturales.

### 9.2.2 Perspectives liées à la mesure de contribution à l'architecture

Notre contribution fait l'objet d'un point d'attention majeur lié au seuil de catégorisation des développeurs emprunté à Bird *et al.* [BNM+11]. Ce seuil a été défini pour mesurer la contribution des développeurs sur des éléments de faible granularité (librairies dll Windows). Or, dans cette thèse, nous l'appliquons à l'ensemble du projet, soit la granularité maximale, avec un projet grandissant potentiellement au fil du temps. De fait, il se peut que le seuil de 5% soit plus sévère appliqué globalement sur le projet que sur une unité du projet.
Une première perspective pourrait donc être de mener une étude sur la pertinence de l'adaptation de ce seuil lorsque la taille d'un projet croît, par exemple considérant un seuil proportionnel à l'inverse de la taille du projet plutôt que fixe.
Une deuxième perspective serait de mesurer le taux de contribution et les catégories de développeurs associées pour d'autres types de contributions ou niveaux architecturaux. Cela permettrait la l'étude des similarités et disparités entre développeurs pour plusieurs types de contributions ou niveaux d'architectures. De ce fait, il serait envisageable d'étudier par exemple les développeurs contribuant au code Java et ceux contribuant à l'architecture *runtime* construite à l'aide du *framework* Spring.

Enfin, une troisième perspective porterait sur l'étude de "points chauds" (*hot spots*) dans le projet. Le taux de propriété architecturale et les catégories de développeurs pourraient être mesurés puis associés aux différents éléments du projet. De cette manière, il serait possible de prédire le nombre de développeurs expérimentés sur l'architecture pour chaque point chaud, en fonction de leur taille. La prédiction du nombre de développeurs expérimentés sur l'architecture pour l'intégralité du projet pourrait, elle aussi, être un indicateur de management intéressant.

### 9.2.3 Perspectives liées à la classification automatique de développeurs

Cette contribution présente certaines limitations liées aux variables utilisées et à l'étiquetage du jeu de données. Nous utilisons 23 variables représentant chacune une métrique logicielle. La qualité du classifieur pourrait être améliorée en utilisant d'autres métriques. L'étiquetage étant basé sur une déclaration des développeurs, il est également possible que des erreurs d'étiquetage existent et influencent la qualité de la classification.
Une première perspective concerne la généricité du classifieur. Nous utilisons un jeu de données avec des variables spécifiques à certaines technologies (Java, *framework* Spring ou encore Maven et Gradle). Une possibilité serait de s'abstraire de ces variables spécifiques. Une autre possibilité pourrait être l'adaptation de ce travail à d'autres langages orientés objets comme C++, en comptant non plus le nombre de lignes ajoutées ou supprimées dans des fichiers Gradle ou Maven, mais plutôt dans des fichiers Makefile.
Une deuxième perspective concerne l'amélioration et l'enrichissement du jeu de données en vue d'obtenir de meilleures performances. Une première piste pourrait être l'ajout des données issues d'autres projets *open-source*. Une seconde piste serait son enrichissement avec des projets en sources fermées provenant d'industriels.

### 9.2.4 Perspectives liées à l'étude des profils de développeurs

Une première perspective concerne la prédiction du nombre de développeurs expérimentés en fonction de la taille du projet ou de ses technologies architecturales. Nous avons vu qu'il existe une corrélation entre la taille du projet et le nombre de développeurs expérimentés. Avec plus de données, il serait intéressant d'explorer ce lien pour la création d'un prédicteur du nombre de développeurs expérimentés nécessaires à un projet donné en se basant sur la taille du projet.
Une deuxième perspective concerne la prédiction de la capacité d'un développeur non-expérimenté à devenir expérimenté. Connaissant les évolutions des caractéristiques (métriques) des développeurs entre les deux catégories, il serait possible de prédire le potentiel d'un développeur non-expérimenté à devenir expérimenté. Tout comme la perspective précédente, cela nécessiterait un volume de données important dont nous ne disposons pas ici.

### 9.2.5 Perspectives expérimentales

Deux contributions principales offrent des perspectives expérimentales.

La première concerne la métrique d'évaluation de la contribution architecturale (Chapitre II.5). Une expérimentation impliquant un nombre plus important de projets de même nature, utilisant le langage Java et le *framework* Spring, permettrait de conduire une étude empirique plus large sur les types de développeurs. De la même façon, il pourrait être intéressant d'étendre le panel de projets en faisant varier la technologie de construction de l'architecture ainsi que le style architectural ou le niveau d'architecture.
La seconde contribution qui ouvre des perspectives expérimentales intéressantes concerne l'étude du profil des développeurs (Chapitre II.7). En utilisant un panel de projets plus grand, nous pourrions consolider les résultats obtenus et étudier les différences de profils entre les types de projets ou d'architectures.

# Publications

Quentin Perez, Pierre-Antoine Jean, Christelle Urtado, Sylvain Vauttier. Bug or not bug? That is the question. ICPC 2021 - 29th IEEE/ACM International Conference on Program Comprehension, May 2021, Virtual. pp.47–58.

Quentin Perez, Alexandre Le Borgne, Christelle Urtado, Sylvain Vauttier. Towards Profiling Runtime Architecture Code Contributors in Software Projects. ENASE 2021 - 16th International conference on Evaluation of Novel Approaches to Software Engineering, Apr 2021, Virtual. pp.429–436.

Quentin Perez, Alexandre Le Borgne, Christelle Urtado, Sylvain Vauttier. An Empirical Study about Software Architecture Configuration Practices with the Java Spring Framework. SEKE 2019 - 31st Software Engineering and Knowledge Engineering, Jul 2019, Lisbonne, Portugal. pp.465–468.

# Bibliographie


[AA05] John Arthur and Shiva Azadegan. Spring framework for rapid open source J2EE web application development: A case study. In *6th ACIS*, pages 90–95, Towson, USA, May 2005. IEEE.

[AAP+08] Giuliano Antoniol, Kamel Ayari, Massimiliano Di Penta, Foutse Khomh, and Yann-Gaël Guéhéneuc. Is it a bug or an enhancement?: a text-based approach to classify change requests. In *Proceedings of the 2008 conference of the Centre for Advanced Studies on Collaborative Research*, page 23, Richmond Hill, Canada, October 2008.

[ABT+18] Mauricio Finavaro Aniche, Gabriele Bavota, Christoph Treude, Marco Aurélio Gerosa, and Arie van Deursen. Code smells for model-view-controller architectures. *Empirical Software Engineering*, 23(4):2121–2157, 2018.

[AG12] Anne Archambault and Jonathan Grudin. A longitudinal study of facebook, linkedin, & twitter use. In Joseph A. Konstan, Ed H. Chi, and Kristina Höök, editors, *30th CHI*, pages 2741–2750, Austin, USA, May 2012. ACM.

[AGE95] F Brito Abreu, Miguel Goulão, and Rita Esteves. Toward the design quality evaluation of object-oriented software systems. In *Proceedings of the 5th ICSQ, Austin, Texas, USA*, pages 44–57, Austin, USA, 1995.

[AHM05] John Anvik, Lyndon Hiew, and Gail C. Murphy. Coping with an open bug repository. In *OOPSLA workshop on Eclipse Technology eXchange*, pages 35–39, San Diego, USA, October 2005. ACM.

[AMP+19] Mathieu Acher, Hugo Martin, Juliana Alves Pereira, Arnaud Blouin, Jean-Marc Jézéquel, Djamel Eddine Khelladi, Luc Lesoil, and Olivier Barais. Learning Very Large Configuration Spaces: What Matters for Linux Kernel Sizes. Research report, October 2019.

[BB10] Adrian Bachmann and Abraham Bernstein. When process data quality affects the number of bugs: Correlations in software engineering datasets. In *7th International Working Conference on Mining Software Repositories, MSR*, pages 62–71, Cape Town, South Africa, May 2010. IEEE.

[Bec99] Kent L. Beck. Embracing change with extreme programming. *Computer*, 32(10):70–77, 1999.

[Ber08] Brian Berenbach. The other skills of the software architect. In *1st LMSA*, page 7–12, New York, USA, 2008. Association for Computing Machinery.

[BFB99] Kent Beck, Martin Fowler, and Grandma Beck. Bad smells in code. *Refactoring: Improving the design of existing code*, 1(1999):75–88, 1999.

[BJC+13] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. Reversible debugging software. *Judge Business School Technical Report*, 2013.

[BLB+13] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the Scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

[BM70] Marc Barbut and Bernard Monjardet. *Ordres et classifications : Algèbre et combinatoire, vols. 1 and 2.* Hachette, Paris, 1970.

[BNM+11] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. Don't touch my code! examining the effects of ownership on software quality. In *19th ACM SIGSOFT FSE*, pages 4–14, Szeged, Hungary, Sept. 2011. ACM.

[Boo96] Grady Booch. *Object solutions: managing the object-oriented project.* Addison-Wesley, 1996.

[CBHK02] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. SMOTE: synthetic minority over-sampling technique. *Journal Of Artificial Intelligence Research*, 16:321–357, 2002.

[CHT00] Colleen Cook, Fred Heath, and Russel L Thompson. A meta-analysis of response rates in web-or internet-based surveys. *Educational and psychological measurement*, 60(6):821–836, 2000.

[CK94] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[CS15] Indu Chawla and Sandeep K. Singh. An automated approach for bug categorization using fuzzy logic. In Srinivas Padmanabhuni, Raghu Nambiar, Premkumar T. Devanbu, Murali Krishna Ramanathan, and Ashish Sureka, editors, *8th India Software Engineering Conference*, pages 90–99, Bangalore, India, February 2015. ACM.

[CZ17] Roberto Di Cosmo and Stefano Zacchiroli. Software heritage: Why and how to preserve software source code. In *4th iPRES*, Kyoto, Japan, September 2017.

[Dar09] Charles Darwin. *The origin of species.* PF Collier & son New York, 1909.

[DBSS13] Enrico Di Bella, Alberto Sillitti, and Giancarlo Succi. A multivariate classification of open source developers. *Information Sciences*, 221:72–83, 2013.

[DKD21] Malinda Dilhara, Ameya Ketkar, and Danny Dig. Understanding software-2.0: A study of machine learning library usage and evolution. *ACM Transactions on Software Engineering and Methodology*, 30(4):55:1–55:42, 2021.

[DM20] Mario Dudjak and Goran Martinović. In-depth performance analysis of smote-based oversampling algorithms in binary classification. *International Journal of Electrical and Computer Engineering Systems*, 11(1):13–23, 2020.

[EL16] Michael D. Ekstrand and Michael Ludwig. Dependency injection with static analysis and context-aware policy. *Journal of Object Technology*, 15(1):1:1–31, February 2016.

[Eri06] K. Anders Ericsson. *An Introduction to The Cambridge Handbook of Expertise and Expert Performance: Its Development, Organization, and Content*, page 3–20. Cambridge Handbooks in Psychology. Cambridge University Press, 2006.

[FFB14] Matthieu Foucault, Jean-Rémy Falleri, and Xavier Blanc. Code ownership in open-source software. In *18th EASE*, pages 1–9, London, UK, May 2014. ACM.

[FH+01] Martin Fowler, Jim Highsmith, et al. The agile manifesto. *Software Development*, 9(8):28–35, 2001.

[FMZM16] Francesca Arcelli Fontana, Mika V Mäntylä, Marco Zanoni, and Alessandro Marino. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, 21(3):1143–1191, 2016.

[Fow04] Martin Fowler. Inversion of control containers and the dependency injection pattern. Online article, January 2004. https://martinfowler.com/articles/injection.html [2021-07-10].

[FPB+15] Matthieu Foucault, Marc Palyart, Xavier Blanc, Gail C Murphy, and Jean-Rémy Falleri. Impact of developer turnover on quality in open-source software. In *10th ESEC / FSE*, pages 829–841, Bergamo, Italy, Sept. 2015. ACM.

[FTL+15] Matthieu Foucault, Cédric Teyton, David Lo, Xavier Blanc, and Jean-Rémy Falleri. On the usefulness of ownership metrics in open-source software projects. *Inf. Softw. Technol.*, 64:102–112, 2015.

[GF16] Gillian J Greene and Bernd Fischer. CVExplorer: Identifying candidate developers by mining and exploring their open source contributions. In *31st IEEE/ACM ASE*, pages 804–809, Singapore, Singapore, 2016. ACM.

[Gir21] Görkem Giray. A software engineering perspective on engineering machine learning systems: State of the art and challenges. *Journal of Systems and Software*, 180:111031, 2021.

[Gon08] Iker Gondra. Applying machine learning to software fault-proneness prediction. *Journal of Systems and Software*, 81(2):186–195, 2008.

[GR12] Jesús M. González-Barahona and Gregorio Robles. On the reproducibility of empirical software engineering studies based on data retrieved from development repositories. *Empirical Software Engineering*, 17(1-2):75–89, 2012.

[GRD97] M. Gen, Runwei Cheng, and Dingwei Wang. Genetic algorithms for solving shortest path problems. In *IEEE International Conference on Evolutionary Computation*, pages 401–406, Indianapolis, USA, 1997.

[GS93] David Garlan and Mary Shaw. An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering*, volume 2 of *Series on Software Engineering and Knowledge Engineering*, pages 1–39. World Scientific, 1993.

[GW99] Bernhard Ganter and Rudolf Wille. *Formal concept analysis, Mathematical foundations*. Springer Verlag, Berlin, 1999.

[Hal77] Maurice H Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science, 1977.

[HG15] Claudia Hauff and Georgios Gousios. Matching GitHub developer profiles to job advertisements. In *12th MSR*, pages 362–366, Florence, Italy, May 2015. IEEE.

[HJ21] Gordon Hogenson and Mike Jones. Inversion of control containers and the dependency injection pattern. Online article, January 2021. https://docs.microsoft.com/en-us/visualstudio/code-quality/code-metrics-maintainability-index-range-and-meaning?view=vs-2019 [2021-07-20].

[HJZ13] Kim Herzig, Sascha Just, and Andreas Zeller. It's not a bug, it's a feature: how misclassification impacts bug prediction. In David Notkin, Betty H. C. Cheng, and Klaus Pohl, editors, *35th International Conference on Software Engineering*, pages 392–401, San Francisco, USA, May 2013. IEEE.

[HNA+17] Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, and Patrick Heymans. Yo variability! jhipster: a playground for web-apps analyses. In *11th International Workshop on Variability Modelling of Software-intensive Systems*, pages 44–51, Eindhoven, Netherlands, February 2017. ACM.

[HNA+19] Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, and Benoit Baudry. Test them all, is it worth it? assessing configuration sampling on the jhipster web development stack. *Empirical Software Engineering*, 24(2):674–717, 2019.

[Hol92] John H. Holland. Genetic algorithms. *Scientific American*, 267(1):66–73, 1992.

[ICROGB09] Daniel Izquierdo-Cortazar, Gregorio Robles, Felipe Ortega, and Jesus M Gonzalez-Barahona. Using software archaeology to measure knowledge loss in software projects due to developer turnover. In *42nd HICSS*, page 10, Waikoloa, USA, Jan. 2009. IEEE.

[IROG09] Daniel Izquierdo-Cortazar, Gregorio Robles, Felipe Ortega, and Jesús M. González-Barahona. Using software archaeology to measure knowledge loss in software projects due to developer turnover. In *42st HICSS*, pages 1–10, Waikoloa, USA, 2009. IEEE.

[JKZ09] Gaeul Jeong, Sunghun Kim, and Thomas Zimmermann. Improving bug triage with bug tossing graphs. In *7th ESEC/FSE*, pages 111–120, Amsterdam, The Netherlands, August 2009. ACM.

[Jon72] Karen Sparck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, 1972.

[JS89] Kenneth A. De Jong and William M. Spears. Using genetic algorithms to solve NP-complete problems. In J. David Schaffer, editor, *3rd International Conference on Genetic Algorithms*, pages 124–132, Fairfax, USA, June 1989. Morgan Kaufmann.

[JTPA81] Robin Jeffries, Althea A Turner, Peter G Polson, and Michael E Atwood. The processes involved in designing software. *Cognitive skills and their acquisition*, 255:283, 1981.

[KDCP21] Rafael Kallis, Andrea Di Sorbo, Gerardo Canfora, and Sebastiano Panichella. Predicting issue types on GitHub. *Science of Computer Programming*, 205:102598, 2021.

[KGB+16] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M Germán, and Daniela E Damian. An in-depth study of the promises and perils of mining GitHub. *ESE*, 21(5):2035–2071, 2016.

[KHM08] Huzefa H. Kagdi, Maen Hammad, and Jonathan I. Maletic. Who can help me with this source code change? In *24th ICSM*, pages 157–166, Beijing, China, 2008. IEEE Computer Society.

[KMC06] A. J. Ko, Brad A. Myers, and Duen Horng Chau. A linguistic analysis of how people describe software problems. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 127–134, Brighton, UK, September 2006. IEEE.

[Kru99] Philippe Kruchten. The software architect. In *1st WICSA*, volume 140 of *IFIP Conference Proceedings*, pages 565–584, San Antonio USA, 1999. Kluwer.

[KSCP19] Rafael Kallis, Andrea Di Sorbo, Gerardo Canfora, and Sebastiano Panichella. Ticket Tagger: Machine learning driven issue classification. In *35th ICSME*, pages 406–409, Cleveland, USA, September 2019. IEEE.

[LB20] Alexandre Le Borgne. *ARIANE : Automated Re-Documentation to Improve software Architecture uNderstanding and Evolution.* PhD thesis, IMT - MINES ALES - IMT - Mines Alès Ecole Mines - Télécom, January 2020.

[LDB17] Felix Last, Georgios Douzas, and Fernando Bacao. Oversampling for imbalanced learning based on k-means and smote. 2017.

[LHM14] Nachai Limsettho, Hideaki Hata, and Ken-ichi Matsumoto. Comparing hierarchical dirichlet process with latent dirichlet allocation in bug report multiclass classification. In *15th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, pages 1–6, Las Vegas, USA, June 2014. IEEE.

[LL17] Scott M. Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In *30th NIPS*, pages 4765–4774, Long Beach, USA, 2017.

[LSK+19] Bancha Luaphol, Boonchoo Srikudkao, Tontrakant Kachai, Natthakit Srikanjanapert, Jantima Polpinij, and Poramin Bheganan. Feature comparison for automatic bug report classification. In *15th IC2IT*, pages 69–78, Bangkok, Thailand, 2019. Springer.

[Map16] Simon Maple. Java tools and technologies landscape report. Online article, 2016. https://zeroturnaround.com/rebellabs/java-tools-and-technologies-landscape-2016/ [accessed 2019-03-08].

[Mat96] Michael Mattsson. Object-oriented frameworks. *Licentiate thesis*, 1996.

[McB07] Matthew R. McBride. The software architect. *Communication ACM*, 50(5):75–81, May 2007.

[McC76] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

[MFH02] Audris Mockus, Roy T Fielding, and James D Herbsleb. Two case studies of open source software development: Apache and Mozilla. *ACM TOSEM*, 11(3):309–346, 2002.

[MH02] Audris Mockus and James D Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *24th ICSE*, pages 503–512, Orlando, USA, May 2002. ACM.

[MHU+15] Abderrahman Mokni, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and Huaxi (Yulin) Zhang. An evolution management model for multi-level component-based software architectures. In Haiping Xu, editor, *27th SEKE*, pages 674–679, Pittsburgh, USA, July 2015. KSI Research Incorporation.

[MNK03] Robert C Martin, James Newkirk, and Robert S Koss. *Agile software development: principles, patterns, and practices*, volume 2. Prentice Hall Upper Saddle River, 2003.

[MZS+18] Lei Ma, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Felix Juefei-Xu, Chao Xie, Li Li, Yang Liu, Jianjun Zhao, and Yadong Wang. Deepmutation: Mutation testing of deep learning systems. In *29th ISSRE*, pages 100–111, Memphis, USA, 2018. IEEE Computer Society.

[NYN+02] Kumiyo Nakakoji, Yasuhiro Yamamoto, Yoshiyuki Nishinaka, Kouichi Kishida, and Yunwen Ye. Evolution patterns of open-source software systems and communities. In *5th IWPSE @ 24th ICSE*, pages 76–85, Orlando, USA, May 2002. ACM.

[OAH19] Ahmed Fawzi Otoom, Sara Al-jdaeh, and Maen Hammad. Automated classification of software bug reports. In *9th ICICM*, pages 17–21, Prague, Czech Republic, August 2019. ACM.

[OBA17] Babatunde Kazeem Olorisade, Pearl Brereton, and Peter Andras. Reproducibility of studies on text mining for citation screening in systematic reviews: Evaluation and checklist. *Journal of Biomedical Informatics*, 73:1–13, 2017.

[OH92] Paul W. Oman and Jack R. Hagemeister. Metrics for assessing a software system's maintainability. In *1st ICSM*, pages 337–344, Orlando, USA, November 1992. IEEE Computer Society.

[PHM13] Natthakul Pingclasai, Hideaki Hata, and Ken-ichi Matsumoto. Classifying bug reports to bugs and other requests using topic modeling. In Pornsiri Muenchaisri and Gregg Rothermel, editors, *20th Asia-Pacific Software Engineering Conference*, volume 2, pages 13–18, Bangkok, Thailand, December 2013. IEEE.

[PLBUV19] Quentin Perez, Alexandre Le Borgne, Christelle Urtado, and Sylvain Vauttier. An empirical study about software architecture configuration practices with the java spring framework. In *31st SEKE*, pages 465–593, Lisbon, Portugal, July 2019. KSI Research.

[PMBF19] João Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. A large-scale study about quality and reproducibility of jupyter notebooks. In *6th ICMSR, MSR*, pages 507–517, Montreal, Canada, May 2019. IEEE / ACM.

[Pre97] Wolfgang Pree. Component-based software development-a new paradigm in software engineering? In *4th APSEC*, pages 523–524. IEEE Computer Society, 1997.

[PSHS17] Nitish Pandey, Debarshi Kumar Sanyal, Abir Hudait, and Amitava Sen. Automated classification of software issue reports using machine learning techniques: an empirical study. *Innovations in Systems and Software Engineering*, 13(4):279–297, 2017.

[QS18] Hanmin Qin and Xin Sun. Classifying bug reports into bugs and non-bugs using LSTM. In *10th Asia-Pacific Symposium on Internetware*, pages 20:1–20:4, Beijing, China, September 2018. ACM.

[RFPZ18] Riccardo Roveda, Francesca Arcelli Fontana, Ilaria Pigazzini, and Marco Zanoni. Towards an architectural debt index. In *44th SEAA*, pages 408–416, Prague, Czech Republic, August 2018. IEEE Computer Society.

[RG+11] Noraini Mohd Razali, John Geraghty, et al. Genetic algorithm performance with different selection strategies in solving tsp. In *World congress on engineering*, volume 2, pages 1–6, London, UK, 2011. International Association of Engineers Hong Kong.

[RGB06] Gregorio Robles and Jesus M Gonzalez-Barahona. Contributor turnover in libre software projects. In *2nd IFIP OSS*, pages 273–286, Como, Italy, June 2006. Springer.

[RN20] Stuart Russell and Peter Norvig. *Artificial intelligence: a modern approach, 4th edition*. Pearson Education Limited, 2020.

[RSG16] Marco Túlio Ribeiro, Sameer Singh, and Carlos Guestrin. "why should I trust you?": Explaining the predictions of any classifier. In *22nd SIGKDD*, pages 1135–1144, San Francisco, USA, 2016. ACM.

[RSG18] Marco Túlio Ribeiro, Sameer Singh, and Carlos Guestrin. Anchors: High-precision model-agnostic explanations. In *32nd AAAI*, pages 1527–1535, New Orleans, USA, 2018. AAAI Press.

[SA91] William M. Spears and Vic Anand. A study of crossover operators in genetic programming. In *Methodologies for Intelligent Systems*, pages 409–418. Springer, 1991.

[SAB18] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. PyDriller: Python framework for mining software repositories. In *26th ESEC/FSE*, New York, USA, November 2018. ACM.

[Sch97] Ken Schwaber. Scrum development process. In *Business object design and implementation*, pages 117–134. Springer, 1997.

[SdASOF18] Adriano Lages Dos Santos, Maurício R. de A. Souza, Johnatan Oliveira, and Eduardo Figueiredo. Mining software repositories to identify library experts. In *7th SBCARS*, pages 83–91, Sao Carlos, Brazil, 2018. ACM.

[SF95] K. Srinivasan and D. Fisher. Machine learning approaches to estimating software development effort. *IEEE Transactions on Software Engineering*, 21(2):126–137, 1995.

[SFHA14] Hassan Saif, Miriam Fernández, Yulan He, and Harith Alani. On stopwords, filtering and data sparsity for sentiment analysis of twitter. In *9th LREC*, pages 810–817, Reykjavik, Iceland, May 2014. European Language Resources Association (ELRA).

[SGM02] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component software: beyond object-oriented programming*. Pearson Education, 2002.

[SH15] Sofia Sherman and Irit Hadar. Toward defining the role of the software architect. In *8th IWCHASE*, pages 71–76, 2015.

[SHN15] Masumi Shirakawa, Takahiro Hara, and Shojiro Nishio. N-gram IDF: A global term weighting scheme based on information distance. In *24th WWW*, pages 960–970, Florence, Italy, May 2015. ACM.

[Sin08] Renuka Sindhgatta. Identifying domain expertise of developers from source code. In *14th ACM SIGKDD KDD*, pages 981–989, Las Vegas, USA, August 2008. ACM.

[SKL+14] Janet Siegmund, Christian Kästner, Jörg Liebig, Sven Apel, and Stefan Hanenberg. Measuring and modeling programming experience. *Empirical Software Engineering*, 19(5):1299–1334, 2014.

[SME+17] Anas Shatnawi, Hafedh Mili, Ghizlane El-Boussaidi, Anis Boubaker, Yann-Gaël Guéhéneuc, Naouel Moha, Jean Privat, and Manel Abdellatif. Analyzing program dependencies in java EE applications. In *14th MSR*, pages 64–74, Buenos Aires, Argentina, May 2017. IEEE.

[SNV06] Sabine Sonnentag, Cornelia Niessen, and Judith Volmer. Expertise in software design. In K. Anders Ericsson, Neil Charness, Paul J. Feltovich, and Robert R. Hoffmann, editors, *Cambridge handbook of expertise and expert performance*, pages 373–387. Cambridge University Press, 2006.

[Som15] Ian Sommerville. *Software engineering 10th Edition*. Pearson, 2015.

[Son95] Sabine Sonnentag. Excellent software professionals: Experience, work activities, and perception by peers. *Behaviour & Information Technology*, 14(5):289–299, 1995.

[Son98] Sabine Sonnentag. Expertise in professional software design: A process study. *Journal of applied psychology*, 83(5):703, 1998.

[SZ08] David Schuler and Thomas Zimmermann. Mining usage expertise from version archives. In *5th MSR*, pages 121–124, Leipzig, Germany, May 2008. ACM.

[TFMB13] Cédric Teyton, Jean-Rémy Falleri, Floréal Morandat, and Xavier Blanc. Find your library experts. In *20th WCRE*, pages 202–211, Koblenz, Germany, 2013. IEEE.

[TG94] D. Thierens and D. Goldberg. Elitist recombination: an integrated selection recombination GA. In *1st conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*, volume 1, pages 508–512, Orlando, USA, 1994.

[TGAJ16] Paul Temple, José Angel Galindo, Mathieu Acher, and Jean-Marc Jézéquel. Using machine learning to infer constraints for product lines. In *20th SPLC*, pages 209–218, Beijing, China, September 2016. ACM.

[THK07] Chouki Tibermacine, Didier Hoareau, and Reda Kadri. Enforcing architecture and deployment constraints of distributed component-based software. In *10th FASE*, volume 4422, pages 140–154, Braga, Portugal, March 2007. Springer.

[THPM17] Pannavat Terdchanakul, Hideaki Hata, Passakorn Phannachitta, and Kenichi Matsumoto. Bug or not? bug report classification using n-gram IDF. In *IEEE International Conference on Software Maintenance and Evolution*, pages 534–538, Shanghai, China, September 2017. IEEE.

[TMD09] R.N. Taylor, N. Medvidovic, and E. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2009.

[TPF+14] Cédric Teyton, Marc Palyart, Jean-Rémy Falleri, Floréal Morandat, and Xavier Blanc. Automatic extraction of developer expertise. In *18th EASE*, page 8, London, UK, May 2014. ACM.

[Tse98] Alan CB Tse. Comparing response rate, response speed and response quality of two methods of sending questionnaires: e-mail vs. mail. *Market Research Society Journal*, 40(4):1–12, 1998.

[TSTD16] Chouki Tibermacine, Salah Sadou, Minh Tu Ton That, and Christophe Dony. Software architecture constraint reuse-by-composition. *Future Gener. Comput. Syst.*, 61:37–53, 2016.

[ULD+20] Mark Utting, Bruno Legeard, Frédéric Dadeau, Frédéric Tamagnan, and Fabrice Bouquet. Identifying and generating missing tests using machine learning on execution traces. In *2nd International Conference On Artificial Intelligence Testing, AITest*, pages 83–90, Oxford, UK, August 2020. IEEE.

[ZC18] Alice Zheng and Amanda Casari. *Feature engineering for machine learning: principles and techniques for data scientists*. O'Reilly, 2018.

[ZTGG16] Yu Zhou, Yanxiang Tong, Ruihang Gu, and Harald C. Gall. Combining text mining and data mining for bug report classification. *Journal of Software Evolution Process*, 28(3):150–176, 2016.

[ZUV10] Huaxi (Yulin) Zhang, Christelle Urtado, and Sylvain Vauttier. Architecture-centric component-based development needs a three-level ADL. In *4th ECSA 2010, Copenhagen*, volume 6285, pages 295–310, Copenhagen, Denmark, August 2010. Springer.

[ZZU+12] Huaxi (Yulin) Zhang, Lei Zhang, Christelle Urtado, Sylvain Vauttier, and Marianne Huchard. A three-level component model in component based software development. In *11th GPCE*, pages 70–79, Dresden,Germany, September 2012. ACM.

**Résumé :** En génie logiciel, l'outillage des processus de développement est aujourd'hui indispensable à la gestion qualitative des projets et peut couvrir différentes phases : conception, codage ou maintenance. Cet outillage protéiforme regroupe les mesures appelées métriques, les systèmes automatisés de production de code et les systèmes d'aide à la décision, prédictifs ou non. Le champ d'application de ces derniers est large : détection de bad smells, classification de tickets, etc. Dans cette thèse, nous nous intéressons à la production de métriques portant sur la conduite des projets logiciels en s'appuyant sur les contributions au code source et plus particulièrement à l'architecture runtime des applications. Nous utilisons également des méta-données liées à l'historique du projet et à son développement. La première problématique étudiée concerne la classification de tickets logiciels décrivant des bogues. Nous entraînons un réseau de neurones sur un jeu de données reconnu par la communauté scientifique et nous optimisons ses hyper-paramètres à l'aide d'un algorithme génétique. De cette manière, nous obtenons des performances de classification supérieures à l'état de l'art. Ces performances permettent d'envisager l'intégration de notre classifieur dans des outils de gestion de projets ou dans un assistant à la rédaction de tickets. La deuxième question abordée est le turn-over des développeurs contribuant au développement de l'architecture logicielle. La détection des développeurs contribuant de manière majeure ou mineure a déjà été formulée et des métriques dédiées proposées. Cependant, les approches existantes ne se focalisent pas spécifiquement sur le développement de l'architecture. C'est pourquoi nous avons créé un modèle formel ainsi qu'une métrique de contribution au code de l'architecture. Nous validons cette métrique de manière empirique puis nous proposons une analyse des différentes catégories de développeurs extraites par notre métrique. Les résultats mettent en lumière la présence de catégories de développeurs spécifiques dont un noyau de développeurs expérimentés contribuant de manière majeure au code de l'architecture durant toute la vie du projet. La détection des développeurs expérimentés ayant de potentielles connaissances en architectures est également un problème que nous étudions. Contrairement aux approches existantes, nous utilisons l'apprentissage supervisé pour apprendre des profils de développeurs expérimentés. Aucun jeu de données étiquetées n'étant disponible dans la littérature scientifique, nous avons créé un jeu de données dédié. Pour cela nous extrayons plusieurs centaines de développeurs issus de 17 projets open source. Nous calculons ensuite 23 métriques pour chacun des développeurs contribuant à ces projets. Enfin, nous étiquetons notre jeu de données manuellement en utilisant les réseaux sociaux professionnels ainsi que la documentation des projets. Nous entraînons ensuite un classifieur sur ce jeu de données et nous l'évaluons. Nos résultats montrent de bonnes performances sur la détection de profils de développeur expérimentés, ce qui permet d'envisager l'utilisation d'un tel outil pour faciliter le management des projets. Enfin, nous réalisons une étude par croisement des résultats obtenus via la métrique de contribution à l'architecture et ceux obtenus par la classification automatique des développeurs. Cette étude permet d'analyser le profil des développeurs expérimentés. Nous analysons également l'évolution des 23 métriques des développeurs durant leur passage d'inexpérimenté à expérimenté. Les résultats montrent qu'une grande part des développeurs expérimentés sont également des contributeurs majeurs à l'architecture et que le changement de catégorie d'un développeur est multifactoriel. Nous fournissons également une évaluation de la reproductibilité de nos travaux en utilisant des cadres méthodologiques définis pour les études en génie logiciel empirique et en apprentissage automatique.

**Mots clés :** Génie logiciel, Apprentissage machine, Métriques, Architecture logicielle, Contribution logicielle

---

**Abstract :** In software engineering, tooling is nowadays mandatory for efficient project management and may relates to any project phase: design, coding or maintenance. These various tools consist of measures, called metrics, automated code generation systems and decision support systems, predictive or not. The applications of the latter are numerous: bad smell detection, ticket classification, etc. This thesis studies the proposal of metrics related to software project management based on contributions to its source code and more specifically to the runtime architecture of applications. Meta-information documenting the history of the project and its development are also used. The automated classification of bug tickets is the first studied question. A neural network is trained with a dataset used as a benchmark in the literature. Its hyper-parameters are then optimized thanks to a genetic algorithm. This way, better classification performances than the state of the art are reached. These performances allow for the integration of our classifier to project management tools or ticket writing assistants. The second addressed issue is the turn-over of the developers who take part to the development of software architectures. Detecting major and minor contributors among developers is an already studied topic and related metrics have been proposed. However, existing approaches do not apply more specifically to architecture development. We have thus designed a formal model and a metrics dedicated to contributions to runtime architecture code. Our metrics has been validated empirically on a real project and the extracted developer categories have been analyzed. Results highlight specific developer categories including a core of expert developers who produce major contributions to the architecture code throughout the history of the project. Detecting expert developers with potential architectural proficiencies is also studied. Unlike existing approaches, supervised machine learning is used to classify the profiles of expert developers. As no labelled dataset is available from the scientific community, we have created a dedicated dataset. For this purpose, hundreds of developer accounts have been extracted from 17 open-source projects. 23 metrics are then calculated for each developer participating to these projects. Finally, the dataset is manually labelled thanks to information from professional social networks and projects' documentation. Good performance on detecting expert developer profiles is achieved, so that this kind of approach could be used to assist project management. Finally, a cross-over study is conducted, combining results from the runtime architecture contribution metrics and results from the expert developer detection. This study enables to analyze the profiles of the retrieved expert developers. We also analyze the evolution of the 23 metrics that compose the profiles when non-expert developers become experts. Results show that a large number of expert developers are also major contributors to the architecture and that developer profile evolution is multifactorial. We also provide an evaluation of reproducibility for our work based on methodological best practices proposed for studies in empirical software engineering and machine learning.

**Keywords :** Software Engineering, Machine Learning, Metrics, Software Architecture, Software contribution