École Polytechnique de Montréal

Département de génie informatique et logiciel

LOG8371
Ingénierie de la qualité en logiciel

TP1- Plan et test

Soumis à Mohamed-Essaddik Benyahia

Soumis par
Julien Arès (1802992)
Olivier Filippelli (1775057)
Kim Thuyen Ton (1789211)
Alex Tran (1799594)

Groupe 03 – Équipe Bravo
Hiver 2019

21 février 2019

Table des matières

Introduction

Le plan d'assurance se porte sur le logiciel Weka. Ce logiciel est open-source. Il est écrit par l'université de Waikato en Nouvelle-Zélande. C'est un outil servant au data mining. Il contient toutes les composantes pour pouvoir utiliser plusieurs algorithmes type de data mining tels que le clustering et la classification et visualiser les données et les résultats. Ce plan se concentre sur deux algorithmes de classification, deux de clustering et un d'association: l'algorithme de bayes naïf, l'algorithme de régression linéaire, l'algorithme de k-mean, l'algorithme de clustering hiérarchique et l'algorithme apriori respectivement.

Des chercheurs utilise ce logiciel et compte sur lui pour avancer leur recherche. Qu'ils soient des chercheurs en intelligence artificielle ou des chercheurs dans d'autres domaines utilisant l'intelligence artificielle durant leur recherche, il est important pour eux que le logiciel ait une bonne qualité. Il est crucial de bien traiter les données pour obtenir des résultats pertinents pour les chercheurs.

Exigences fonctionnelles

Il est important de comprendre ce que les différents algorithmes font pour pouvoir s'assurer de leur qualité. Sans comprendre le comportement attendu, il est impossible de faire une bonne évaluation de la qualité du logiciel.

Algorithme de Bayes Naïf

Cet algorithme classifie les données selon les différents *features* attachés. Cet algorithme se base sur la prémisse que toutes les *features* sont indépendantes l'une de l'autre.

Algorithme de régression linéaire

Cet algorithme essaye de trouver la meilleure droite pour modéliser un ensemble de données selon la relation entre les variables indépendante et dépendante.

Algorithme de k-means

Cet algorithme regroupe les données parmi k regroupements en les regroupant selon leur distance du centre de chaque *cluster*.

Algorithme de clustering hiérarchique

Cet algorithme sépare chaque donnée dans des regroupements de manière hiérarchique. Il existe plusieurs manière de le faire. L'approche agglomérative part avec toutes les données dans leur propre regroupement et les rassemble progressivement selon le critère de distance choisi. L'approche de division commence avec un seul regroupement contenant toutes les données et sépare une donnée à la fois selon le critère de distance choisi.

Algorithme apriori

Cet algorithme associe les données entre elle selon la fréquence d'apparition des données. Pour qu'une association soit accepté, elle doit apparaître au moins autant de fois qu'un seuil prédéterminé parmi les données.

Critères de qualités

Trois critères de qualités sont particulièrement important pour la fonctionnalité, la fiabilité et la maintenabilité. Pour chacun des critères, des sous-critères sont choisi pour mieux définir des objectifs de qualité.

Fonctionnalité

Le critère de fonctionnalité est le critère qui concerne ce que le produit est supposé faire selon les requis prédéterminés. Parmi les sous-critères, les sous-critère de *functional completeness* et de *functional correctness* sont particulièrement importants pour des chercheurs.

Le sous-critère de *functional completeness* vérifie à quel point les fonctions couvrent toutes les tâches déterminées et répondent au besoin des utilisateurs. Ce sous-critère est important, car les chercheurs veulent avoir le plus d'outils à leur disposition pour avancer leur recherche. Plus le logiciel atteint un pourcentage élevé, plus il sera intéressant pour les chercheurs voulant utiliser plusieurs algorithmes.

Le sous-critère de *functional correctness* vérifie à quel point les fonctions retournent les résultats attendus avec la précision voulue. Il est important d'avoir des résultats correctes, car les chercheurs se basent sur elles pour leur recherche. Des mauvais résultats pourraient

engendrer une mauvaise conclusion à l'expérience et induire en erreur la communauté scientifique.

Fiabilité

Le critère de fiabilité est le critère qui observe à quel point le logiciel performe dans différentes conditions contrôlées pour bien identifiés le comportement du logiciel. Parmi les sous-critères de *maturity*, de *fault tolerance* et de *recovery* sont importants à tester pour Weka.

Le sous-critère de *maturity* vérifie à quel point le logiciel répond aux besoins des utilisateurs dans des conditions normales d'utilisation. Ces conditions sont les conditions optimales pour le logiciel pour ne pas causer d'erreurs de fonctionnement. Ce critère est important pour les chercheurs, car il faut qu'ils puissent utiliser la version courante en tout temps. Il ne faut pas que la nouvelle version empêche soudainement l'avancement de leur recherche. De plus, il faudrait que le logiciel offre le plus rapidement possible toutes les algorithmes disponibles de data mining selon l'avancement de la littérature scientifique.

Le sous-critère de *fault tolerance* vérifie à quel point le logiciel peut fonctionner malgré la présence d'erreur de logiciel ou de matériel. Ce sous-critère est important puisque les chercheurs s'attendent à avoir des bons résultats sans corruption. Il faudrait que le logiciel invalide les résultats en cas d'erreurs pour ne pas induire les chercheurs en erreur.

Le sous-critère de *recoverability* vérifie à quel point le logiciel peut retourner à un état normal après une interruption grave. Ce sous-critère est important puisque les chercheurs veulent pouvoir utiliser le logiciel après une interruption grave et ne pas devoir à réinstaller le logiciel.

Maintenabilité

Le critère de maintenabilité est le critère qui observe à quel point le logiciel est facilement maintenable par l'équipe de maintenance. Parmi les sous-critères, les sous-critère d'*analysability* et de *modifiability* sont importants à considérer.

Le sous critère d'*analysability* vérifie à quel point il est facile de comprendre les dépendances des différents modules et ainsi facilement prévoir les impacts sur les anciennes fonctionnalités par les changements ou facilement diagnostiquer le logiciel en cas d'erreurs. Cela est important,

car plus le logiciel est facilement analyser, plus les itérations comportent moins de risques. Cela permet donc aux nouvelles fonctionnalités d'être implémentées plus rapidement. Cela diminue aussi le risque d'introduire des erreurs et facilite l'identifications des causes de ses derniers.

Le sous-critère de *modifiability* vérifie à quel point le logiciel peut être efficacement modifier sans introduire des erreurs ou dégrader des anciennes fonctionnalités. Cela est important, car il ne faut pas que les changements de versions brisent le logiciel et empêche les chercheurs de continuer leurs travaux.

Le sous-critère de *testability* vérifie à quel point le logiciel peut être testé pour obtenir des mesures de qualité. Il est important de se préoccuper de cela, car un logiciel facilement testable a plus de chance de ne pas être défectueux. Cela permet aussi d'être plus confiant envers la qualité du logiciel.

Objectifs, mesures de validation et stratégie de validation

Sous-critère	**Objectifs**	**Mesures de validation**	**Stratégie de validation**
Functional completeness	90-100% des fonctionnalités requises sont présentes	À l'aide des requis, identifier ce qui manque comme fonctionnalités	Tests unitaires
Functional correctness	90-100% des fonctionnalités requises fonctionnent adéquatement	À l'aide des requis, identifier les résultats erronés obtenus après l'exécution du logiciel	Tests unitaires
Maturity	Index de maturité > 0.90	Calcul de l'index de maturité par la formule SMI = Mt - (Fa + Fc + Fd) / Mt	Calcul
Fault tolerance	Le logiciel doit détecter les erreurs 90% du temps	Observer les résultats obtenus lorsqu'il y a des erreurs lors de l'exécution du logiciel	Suite de tests fourni par Weka, tests boite noire.
Recoverability	Le logiciel retourne à l'état stable précédant l'erreur fatale.	Le nombre de fois que le logiciel ne peut pas retourner à un état normal par lui-même	Tests manuels

Analysability	Un score de 4 étoiles selon SIG (système au dessus de la moyenne)	*Unit size*: nombre de lignes dans une méthode 1-15: au moins 57% 16-30: au plus 43% 31-60: au plus 22% 60+: au plus 7%	Analyse du code pour extraire les statistiques, des revues d'inspections pour déterminer le respect des statistiques voulues
		Duplication: redondance: au plus 4.5% non-redondance: au moins 95.5%	
		Unit complexity: nombre de branche dans une méthode (McCabe) 1-5: au moins 75% 6-10: au plus 25% 11-25: au plus10% 25+: au plus 1.5%	
Modifiability	Un score de 4 étoiles selon SIG (système au dessus de la moyenne)	*Duplication*: redondance: au plus 4.5% non-redondance: au moins 95.5%	Analyse du code pour extraire les statistiques, des revues d'inspections pour déterminer le respect des statistiques voulues
		Unit complexity: nombre de branche dans une méthode (McCabe) 1-5: au moins 75% 6-10: au plus 25% 11-25: au plus10% 25+: au plus 1.5%	
		Module coupling: nombre de méthodes appelées provenant d'une classe externe dans une classe 11-20: au plus 20% 21-50: au plus 14% 51+: au plus 6.5%	
Testability	Un score de 4 étoiles selon SIG (système au dessus de la moyenne)	*Unit complexity*: le nombre de branche dans une méthode (McCabe) 1-5: au moins 75% 6-10: au plus 25% 11-25: au plus10% 25+: au plus 1.5%	Analyse du code pour extraire les statistiques, des revues d'inspections pour déterminer le respect des statistiques voulues

Collection de tests

Tableau 1 - Tests selon les sous-critères

Sous-critère	**Test**	**Description du test**	**Scope**
Functional completeness	Suite de tests NaiveBayesTest	Vérifier que les fonctionnalités demandées par les utilisateurs/clients sont bien disponibles dans le logiciel. On assume que la suite de test établie au moment de l'élaboration du plan de test initial représente adéquatement les requis.	Algorithme de Bayes Naïf
Functional correctness	testUpdatingEquality()	Vérifier que le modèle produit est le même lorsqu'il est entraîné de façon incrémentale et lorsqu'il est entraîné avec l'ensemble des données dès le début.	Algorithme / classe
Maturity	Calculer la maturité	SMI = Mt - (Fa + Fc + Fd) / Mt SMI: Indice de maturité logicielle Mt: nombre de modules dans la version courante Fa: nombre de modules ajoutés dans la version courante Fc: nombre de modules changés dans la version courante Fd: nombre de modules supprimés comparativement à l'ancienne version	Logiciel
Fault tolerance	Ensemble de tests statistiques StatsTest	L'ensemble des tests présents dans cette classe	Logiciel
Recoverability	Faire planter l'application	Essayer de faire planter l'application pour étudier le comportement post interruption.	Logiciel
Analysability	Extraire facilement un algorithme de Weka	Enlever un module et vérifier que l'application fonctionne encore.	Logiciel
Modifiability			Logiciel
Testability	Plan de tests	Établir un plan de tests, et effectuer les tests tout au long du développement	Logiciel

Résultats de tests

Le sous-critère *functional completeness* ne peut être bien évalué dans le cadre de ce travail puisque les besoins exacts des utilisateurs ne sont pas disponibles. Toutefois, la figure 1 montre bien que l'algorithme de Baye Naïf est bien de type classifier.

Le sous-critère *correctness* a été testé par le biais des tests unitaires. La figure 1 montre que les tests ont été complétés avec succès.

Le sous-critère *maturity* n'a pas pu être calculée dans le cadre de ce laboratoire puisque les données relatives aux variables Mt, Fa, Fc et Fd n'étaient pas disponibles.

Le sous-critère *fault tolerance* n'a pas été testé. En effet, ce genre de test se fait par des tests boîtes noires par des experts qui comprennent toutes les fonctionnalités et les différents algorithme du logiciel Weka.

Le sous-critère *recoverability* n'a pas pu être testé puisque le programme ne fait que se rouvrir lorsqu'il y a interruption d'une simulation. Cette dernière n'est pas sauvegardée et il faut donc recommencer la simulation de nouveau. Le logiciel Weka met à la disposition de ces usagers un fichier d'historiques des actions effectuées dans le logiciel, qui peut permettre de déterminer la cause de la panne.

Les sous-critères *analysability* et *modifiability* ont été testés simultanément par l'ajout d'un nouvel algorithme (MultilayerPerceptron). Par ailleurs, le sous-critère *analysability* aurait dû être testé selon les propriétés suivantes: le volume, la duplication de code et le balancement des composantes. D'autre part, le sous-critère *modifiability* aurait dû être testé selon les propriétés suivantes: la duplication de code, la complexité de l'unité et l'accouplement de module. Toutefois, par manque d'informations sur l'ensemble du logiciel Weka, les propriétés pour ces sous-critères n'ont pas pu être évaluées que cela soit pour les statistiques ou les revues d'inspection.

Le sous-critère *testability* est testé par le biais des nombreux tests unitaires déjà présents. De plus, il faudrait demander à une équipe d'experts d'analyser tout le code pour déterminer sa complexité par manque de temps pour les statistiques et les revues d'inspections.

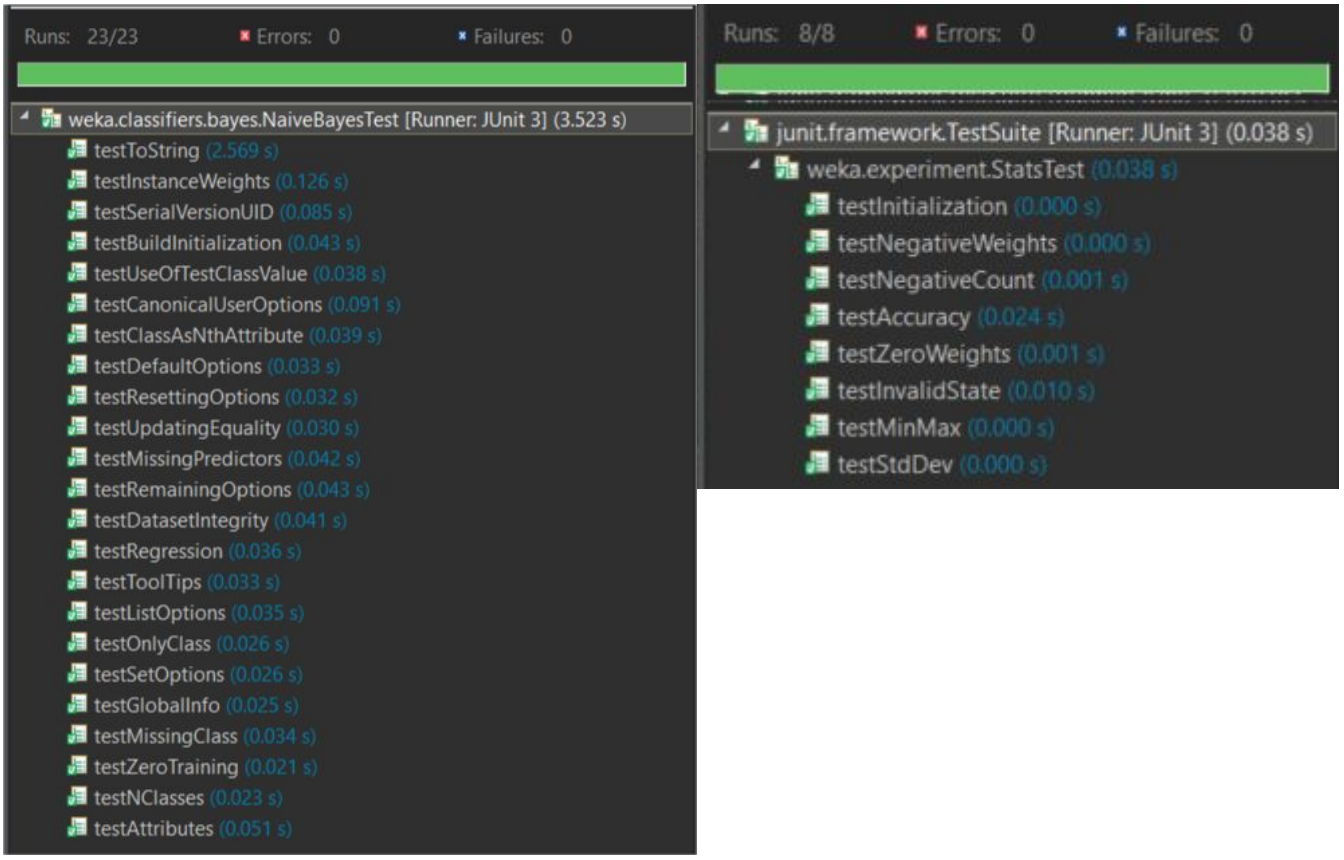

Figure 1 : Résultats des tests unitaires relatifs à l'algorithme de Bayes Naïf

Plan d'intégration continue

Tout d'abord, l'intégration continue est utilisée au cours de développement d'un logiciel pour faire la vérification de l'implémentation du code afin de ne pas produire de régression durant le processus de développement. Cette pratique permet entre autres d'effectuer du développement et de l'intégration en parallèle, et donc de ne pas attendre que tous les modules soient implémentés avant de tester. L'intégration continue consiste alors à effectuer des tests unitaires, des tests d'intégration, des tests de performance, des tests de qualité et plusieurs autres au cours du développement. L'architecture d'un logiciel d'intégration continue est en quelque sorte un gestionnaire de *builds*, de tests et de notifications. Ces éléments vont être représentés sous de plan de processus d'intégration continue:

Avant tout, les développeurs contribuent régulièrement à l'implémentation du logiciel en produisant des *commits*. Le serveur d'intégration continue vérifie les modifications en produisant des *builds* et des tests. Après que les tests soient terminés, une notification est envoyée aux développeurs pour démontrer si l'intégration continue a échoué ou a passé. Si un *build* est échoué, tous les développeurs sont notifiés immédiatement afin que le problème se résolve le plus rapidement possible. C'est ainsi que les développeurs peuvent immédiatement modifier leur code en conséquence pour ne pas engendrer davantage d'erreur dans le logiciel.

Pour notre cas, nous avons utilisé le gestionnaire de développement github pour effectuer les *commits*. Ensuite, notre serveur d'intégration consiste à être une machine virtuelle produite à l'aide de l'outil Travis CI pour vérifier les modifications de code. Le fichier *.travis.yml*, à la racine du projet, permet entre autres d'indiquer à Travis Ci ce qui doit effectuer pour l'intégration continue. On indique alors le chemin permettant d'utiliser l'outil de build *Maven* pour permettre de faire la gestion des dépendances et du déploiement ainsi que *JUnit* pour effectuer les tests. Ainsi, la machine virtuelle *build* et exécute les tests spécifiés dans les fichier *.xml* du code (*build.xml, pom.xml*).

L'intégration continue apporte plusieurs avantages au cours du processus de développement de logiciel. Cela permet entre autres de détecter rapidement les défauts d'implémentation et de les corriger le plus tôt possible dans le cycle de développement. Permets de recevoir rapidement des commentaires sur la qualité et la performance de l'ensemble du système. L'exécution des tests se fait en parallèle avec le développement.

Les tests sont exécutés dans un environnement contrôlé et l'adaptation de ceux-ci se fait facilement.

Nouvel algorithme pour Weka

Le nouvel algorithme qui a été intégré au logiciel Weka est celui MultilayerPerceptron. Celui-ci est en charge des réseaux de neurones et le fichier qui implémente cela se trouve à /weka/src/main/java/weka/classifiers/functions/MultilayerPerceptron.java. Pour garantir la qualité du système, il faudra effectuer les activités d'assurance qualité, notamment en effectuant des tests. Les tests nécessaires doivent vérifier les sous-critères suivants, qui ont été définis dans la première section: "functional completeness", "functional correctness", l'analysabilité, la modifiabilité et la testabilité. De plus, il faudra effectuer des tests d'intégration et de régression. Il n'est pas nécessaire de mettre à jour le plan de qualité puisque celui-ci inclut déjà tous les tests énumérés précédemment. Il faut simplement créer une nouvelle classe de tests pour l'algorithme MultilayerPerceptron qui appelle tous les tests génériques d'un classifier.

Références

[1] Polytechnique Montréal, *Notes de cours*, S.D.

[2] C. Laporte and A. April, *Software Quality Assurance*, IEEE Computer Society, Inc. États-Unis: Wiley, 2018. [En ligne]. Disponible: http://olea.org/recursos/2017-Laporte_April-Software_Quality_Assurance/2017-Laporte,%20April-Software%20Quality%20Assurance,%20First%20Edition-9781119312451.pdf

[3] Orcanos. (2013) Software Maturity Performance. [En ligne]. Disponible: https://www.orcanos.com/compliance/tag/software-maturity/

[4] University of Waikato. (S.D) Weka 3: Data Mining Software in Java. [En ligne]. Disponible : https://www.cs.waikato.ac.nz/ml/weka/