

École Polytechnique de Montréal



LOG8371 - Ingénierie de la qualité en logiciel

Rapport TP1 : Plan d'assurance qualité et test

Equipe Romeo :

Taleb Souli : 1770491

Mehdi Sassi: 1965644

Wassim Khene : 1786511

Abdeltif Ben Brahim: 1709245

21 Février 2019

Table des matières

I. QUESTION 1	1
A. INTRODUCTION ET OBJECTIFS	1
B. LES CRITERES DE QUALITE COUVERTS	2
1. Maintenabilité	2
a) Testabilité	2
b) Modifiabilité	2
2. Fiabilité	2
a) Tolérance aux fautes	2
b) Récupérabilité	3
3. Aptitude fonctionnelle	3
a) Complétude fonctionnelle	3
b) Exactitude fonctionnelle	3
C. LES MESURES ET LES METHODES POUR VALIDER LES OBJECTIFS	3
1. Maintenabilité	3
a) Testabilité	3
b) Modifiabilité	4
2. Fiabilité	4
a) Tolérance aux fautes	4
b) Récupérabilité	4
3. Aptitude fonctionnelle	5
a) Complétude fonctionnelle	5
b) Exactitude fonctionnelle	5
D. LES STRATEGIES DE VALIDATION	5
1. Revues personnelles	5
2. Revues par les paires	5
3. Inspection et Walk-throughs	6
II. QUESTION 2	7
A. PLAN DES TESTS	7
B. DESCRIPTION DES TESTS	7
1. Maintenabilité	7
2. Fiabilité	8
3. Aptitude fonctionnelle	8
C. RAPPORT DES TESTS	8
1. Maintenabilité	8
a) Testabilité	8
b) Modifiabilité	11
2. Fiabilité	12
a) Tolérance aux fautes	12
b) Récupérabilité	13
3. Aptitude fonctionnelle	15
a) Complétude fonctionnelle	15
b) Exactitude fonctionnelle	15
III. QUESTION 3: PLAN D'INTEGRATION CONTINUE	17
IV. QUESTION 4	19
A. « NOUVEAU » ALGORITHME POUR WEKA	19
B. GARANTIR LA QUALITE DU SYSTEME APRES LES MODIFICATIONS	19
C. LES TESTS NECESSAIRES	20
D. LA MISE A JOUR DU PLAN DE QUALITE	20
V. QUESTION 5 : VIDEO	21
ANNEXE	23

Liste des Figures

FIGURE 1: RESULTAT DES TESTS UNITAIRES OBTENUS AVEC JUNIT	9
FIGURE 2: LISTE DES CAS DE TESTS QUI ECHOUENT	9
FIGURE 3: COUVERTURE DU CODE TOTALE DE L'ENSEMBLE DE WEKA.....	10
FIGURE 4: COUVERTURE DU CODE DANS LES ALGORITHMES DE CLASSIFICATION	10
FIGURE 5: COUVERTURE DU CODE DANS LES ALGORITHMES DE <i>CLUSTERING</i>	10
FIGURE 6: COUVERTURE DU CODE DANS L'ALGORITHME D'ASSOCIATION	10
FIGURE 7: COUVERTURE QUI MONTRE LE BON FONCTIONNEMENT DES TESTS DE REGRESSION	11
FIGURE 8: EXEMPLE DE TEST <code>TESTMissingPredictors</code> ET <code>TESTZeroTraining</code>	12
FIGURE 9: EXECUTION DE WEKA ET OUVERTURE D'UN FICHIER *.ARFF	13
FIGURE 10: CHANGEMENTS APPORTES AUX DONNEES	14
FIGURE 11: FORCER LA FERMETURE DE WEKA	14
FIGURE 12: REEXECUTION DE WEKA.....	15
FIGURE 13: LANCEMENT DE JENKINS.....	17
FIGURE 14: CONFIGURATION DE "ANT"	17
FIGURE 15: CHOIX DE DECLENCHEUR DU BUILD	17
FIGURE 16: CONFIGURER LE PROJET GITHUB.....	17
FIGURE 17: RESULTAT DU JOB JENKINS	18
FIGURE 18: NOUVELLE FONCTIONNALITE « <code>NewFunction</code> »	19
FIGURE 19: TEST UNITAIRE AJOUTE POUR TESTER LA NOUVELLE FONCTIONNALITE	19
FIGURE 20: LIGNE DE CODE AJOUTE DANS LE FICHIER TEST « <code>WEKA.AllTests</code> »	20
ANNEXE_FIGURE 1: COUVREGE DE L'ALGORITHME « <code>NaiveBayes</code> »	23
ANNEXE_FIGURE 2: COUVREGE DE L'ALGORITHME « <code>J48</code> »	24
ANNEXE_FIGURE 3: COUVREGE DE L'ALGORITHME « <code>SimpleKMeans</code> »	25
ANNEXE_FIGURE 4: COUVREGE DE L'ALGORITHME « <code>EM</code> »	26
ANNEXE_FIGURE 5: COUVREGE DE L'ALGORITHME « <code>Apriori</code> »	27

Liste des tableaux

TABEAU 1: MESURES DE VALIDATION DE TESTABILITE	4
TABEAU 2: MESURES DE VALIDATION DE MODIFIABILITE	4
TABEAU 3: MESURES DE VALIDATION DE TOLERANCE AUX FAUTES	4
TABEAU 4: MESURES DE VALIDATION DE LA RECUPERABILITE	4
TABEAU 5: MESURES DE VALIDATION DE LA COMPLETUE FONCTIONNELLE	5
TABEAU 6: MESURES DE VALIDATION DE LA CORRECTION FONCTIONNELLE	5
TABEAU 7: OUTIL ET OBJECTIFS DE CHAQUE SOUS-CRITERES D'ASSURANCE QUALITE	7
TABEAU 8: DETAILS DES REVISIONS.....	20

I. Question 1

A. INTRODUCTION ET OBJECTIFS

Ce document couvre le plan de qualité de l'application "Weka". Le plan se focalise sur trois critères de qualité: Aptitude fonctionnelle, Fiabilité et Maintenabilité avec lesquels on va décortiquer la qualité de l'application en se basant sur les objectifs, les mesures et les tests.

Weka est un logiciel d'exploration de données qui regroupe plusieurs algorithmes d'apprentissage automatique pouvant être appliqués directement à un ensemble de données afin de les analyser. Il possède une interface graphique qui permet d'utiliser les différents types d'algorithmes (*regression, pre-processing, clustering, classification*) sur les données entrées.

On accorde une grande importance à la qualité de Weka, étant donné qu'il s'agit d'un programme qui offre plusieurs fonctionnalités en lien avec l'apprentissage automatique et qu'il peut être utilisé dans plusieurs domaines d'application. Pour ces raisons, on cite les avantages de la qualité de ce logiciel :

- Weka doit proposer les fonctionnalités mentionnées dans sa liste d'exigences fonctionnelles. On doit donc s'assurer de la complétude de ses fonctionnalités.
- Weka doit fournir un niveau d'exactitude et de précision acceptable sur les résultats fournis car une erreur de calcul dans Weka entraîne des résultats faussés.
- On doit détecter les problèmes de Weka afin de faciliter la proposition des corrections nécessaires.

Finalement, l'aboutissement du projet Weka ne dépend pas uniquement de la performance de l'équipe de projet. Même si l'équipe représente une partie prenante importante, elle n'est pas la seule. Les parties prenantes peuvent différer en fonction de la nature du projet. Partant de ce fait, on cite comme parties prenantes du projet Weka, l'université de Waikato, la communauté de l'intelligence artificielle et de l'apprentissage automatique et la communauté qui contribue au développement des nouvelles fonctionnalités du projet open source Weka. Entre autres, les chercheurs et les ingénieurs dans le domaine d'intelligence artificielle et d'autres logiciels d'intelligence artificielle qui intègrent weka parmi leurs fonctionnalités tel que KNIME¹ (*Konstanz Information Miner*) et MOA² (*Massive Online Analysis*).

¹ <https://en.wikipedia.org/wiki/KNIME>

² https://en.wikipedia.org/wiki/Massive_Online_Analysis

B. LES CRITERES DE QUALITE COUVERTS

1. Maintenabilité

La maintenabilité est mesurée par le degré d'efficacité et d'efficience avec lequel un produit ou un système peut être modifié par les mainteneurs prévus. Il définit aussi la capacité des applications à être maintenues, de manière cohérente et à moindre coût.

a) Testabilité

Définition : Degré d'efficacité avec lequel les critères de test peuvent être établis pour un système, un produit ou un composant, et des tests pouvant être effectués pour déterminer si ces critères ont été remplis.

Objectif : Weka est un programme qui définit un grand nombre de fonctionnalités avec différents types de données et plusieurs formes de résultats. C'est pour cela que Weka nécessite un jeu de tests varié pour couvrir le maximum de ces fonctionnalités.

b) Modifiabilité

Définition : Mesure dans laquelle un produit ou un système peut être efficacement modifié sans introduire de défauts ni dégrader la qualité des produits existants.

Objectif : Weka est un logiciel qui travaille sur des technologies qui évoluent rapidement, ce qui va entraîner des changements cycliques dans la logique des algorithmes implémentés. De ce fait, la modification introduite sur Weka doit être sans effet sur la totalité du programme ainsi que sur les résultats fournis par Weka.

2. Fiabilité

La fiabilité représente la mesure dans laquelle un système, un produit ou un composant exécute des fonctions précises dans des conditions spécifiées pendant une période donnée.

a) Tolérance aux fautes

Définition : Mesure dans laquelle un système, un produit ou un composant fonctionne comme prévu malgré la présence de pannes matérielles ou logicielles.

Objectif : Weka contient plusieurs fonctionnalités telles que la classification et la régression ce qui nécessite une haute aptitude du système à accomplir les tâches avec le moindre défaut logiciel possible et avec une précision acceptable.

b) Récupérabilité

Définition : Mesure dans laquelle, en cas d'interruption ou de panne, un produit ou un système peut récupérer les données directement affectées et rétablir l'état souhaité du système.

Objectif : Weka est une collection d'algorithmes d'apprentissage automatique pour les tâches d'exploration de données, ce qui fait que les données sont de grande importance. Ainsi, weka doit assurer la récupération des données avec la moindre perte de données possible surtout en cas de panne.

3. Aptitude fonctionnelle

L'Aptitude fonctionnelle est la mesure dans laquelle un produit ou un système fournit des fonctionnalités répondants aux besoins déclarés et implicites lorsqu'il est utilisé dans des conditions spécifiées.

a) Complétude fonctionnelle

Définition : Degré auquel l'ensemble des fonctions couvrent toutes les tâches spécifiées et les objectifs de l'utilisateur.

Objectif : Weka fournit une pléthore de fonctionnalités intéressantes. La complétude fonctionnelle va donc garantir au client la réalisation de toutes les tâches et l'atteinte des objectifs attendus

b) Exactitude fonctionnelle

Définition : Mesure dans laquelle un produit ou un système fournit les résultats exacts avec le degré de précision requis.

Objectif : Weka contient des fonctionnalités pour la préparation des données, la mise en cluster, l'exploration de règles d'association et la visualisation. A cet effet, l'exactitude des résultats fournis est un élément clé à la réussite du programme. En effet, une petite erreur peut conduire le client vers un résultat erroné.

C. LES MESURES ET LES METHODES POUR VALIDER LES OBJECTIFS

1. Maintenabilité

a) Testabilité

Métriques	Critères de réalisation
Nombre de tests unitaires	Il faut fournir au moins un test pour chaque méthode implémentée.
Nombre d'erreurs fatales	Le logiciel ne doit contenir aucune erreur fatale qui bloque le bon fonctionnement du Weka.

Tableau 1: Mesures de validation de testabilité

b) Modifiabilité

Métriques	Critères de réalisation
Nombre de défaillances du système	Notre système doit répondre aux modifications sans entraîner de défaillances système, c'est à dire sans avoir des bugs qui interrompent l'exécution du système.
Nombre de fautes	Le code ne doit contenir aucune erreur qui gênerait le bon fonctionnement du logiciel Weka après avoir ajouté une nouvelle fonctionnalité.

Tableau 2: Mesures de validation de modifiabilité

2. Fiabilité

a) Tolérance aux fautes

Métriques	Critères de réalisation
Nombre de fautes	Il faut définir un nombre maximal de fautes tolérés.
Taille fonctionnelle du produit	Le nombre de lignes de code par fonctionnalité ne doit pas dépasser 10 lignes. Ceci pourrait être fait en se utilisant FPA (<i>Function Point Analysis</i>) et FFP (<i>Full Function Point</i>).

Tableau 3: Mesures de validation de tolérance aux fautes

b) Récupérabilité

Métriques	Critères de réalisation
Temps de récupération	Le logiciel ne doit pas prendre trop de temps pour récupérer ses données.
Nombre d'opérations	Le logiciel doit être capable de reprendre l'exécution d'une activité après une interruption.

Tableau 4: Mesures de validation de la récupérabilité

3. Aptitude fonctionnelle

a) Complétude fonctionnelle

Métriques	Critères de réalisation
Nombre de tâches	Pour accomplir une tâche il faut avoir le minimum d'opérations possible.
Nombre de cas d'utilisation	Avoir un cas d'utilisation pour chaque fonctionnalité fournie.

Tableau 5: Mesures de validation de la complétude fonctionnelle

b) Exactitude fonctionnelle

Métriques	Critères de réalisation
Nombre d'erreurs	Weka doit garantir une marge d'erreur minimale entre les valeurs attendues et les valeurs obtenues.
Nombre d'échecs	Weka doit être capable d'accomplir l'exécution des fonctions sous des limites spécifiques.

Tableau 6: Mesures de validation de la correction fonctionnelle

D. LES STRATEGIES DE VALIDATION

1. Revues personnelles

Les développeurs qui travaillent sur Weka doivent assurer une revue personnelle sur le code qu'ils fournissent, comme un mécanisme d'autocorrection. Partant de ce fait, il faut suivre un processus de revue structuré et prendre la responsabilité de trouver la plupart des défaillances du système (surtout les bogues critiques). Afin de contrôler les revues fournies, chaque développeur doit remplir une liste de contrôle qui définit les tâches fonctionnelles et les tâches incomplètes et/ou manquantes.

2. Revues par les paires

Afin de prévenir plus de problèmes dans weka, on peut appliquer le principe de revue par les paires. Cela consiste à échanger le travail réalisé entre les développeurs et/ou les évaluateurs pour qu'ils l'évaluent et le critiquent. Chaque membre qui fait la revue de l'autre doit fournir une liste de contrôle qui détermine le niveau d'acceptabilité du travail et il doit fournir une note (0, 1 ou 2) pour chaque fonctionnalité. Ces notes signifient :

- 0 : Il faut refaire la partie évalué.
- 1 : Acceptable mais peut être optimisé.
- 2 : Acceptable et optimisé.

3. Inspection et Walk-throughs

En vue de détecter le plus de problème et de fautes critiques au niveau du développement de l'application Weka, il faut organiser des contrôles de code pour s'assurer qu'une évaluation par les pairs est effectuée pour le code sous-jacent. L'équipe de gestion de projet Weka s'assurera que le processus est vérifiable, tandis que l'équipe d'assurance qualité s'assurera que tous les éléments ont été traités et établis. Cela est en partie assuré via le processus de revue de code à travers les *pulls request* et la validation du code sur github puisqu'il s'agit d'un code source ouvert.

II. Question 2

A. PLAN DES TESTS

Critère	Sous-critère	Objectifs	Outils
Maintenabilité	Testabilité	+ Il faut avoir un nombre de tests unitaires qui garantissent un minimum de 70% de couverture des méthodes testées. + Le nombre d'erreurs ne doit pas dépasser 5 au total.	Tests unitaires.
	Modifiabilité	Il faut avoir zéro défaillances du système lors d'une mise à jour du logiciel.	Tests de régression.
Fiabilité	Tolérance aux fautes	Il ne faut pas dépasser 5 fautes totales dans le système Weka.	Test d'acceptation de fautes
	Récupérabilité	Il faut avoir l'option de restaurer la session précédente en cas de panne.	
Aptitude fonctionnelle	Complétude fonctionnelle	Il faut qu'au moins 90% des fonctionnalités implémentées soient compatibles avec les fonctionnalités décrites en SRS (<i>Software Requirements Specification</i>).	Squash TA.
	Exactitude fonctionnelle	Chaque fonctionnalité doit retourner les résultats attendus en fonction de l'entrée qu'elle a reçue.	Tests unitaires

Tableau 7: outil et objectifs de chaque sous-critères d'assurance qualité

B. DESCRIPTION DES TESTS

1. Maintenabilité

Un des sous-critères de la maintenabilité que nous avons choisi est la modifiabilité. On constate que Weka ne propose pas de solutions pour tester la modifiabilité.

Les stratégies les plus pertinentes pour tester la modifiabilité au sein d'un logiciel qui évolue dynamiquement serait d'adopter une stratégie de revue sur plusieurs niveaux (voir question 1). On peut utiliser un outil d'intégration continue comme Jenkins (voir question 3) qui va tester automatiquement la capacité du système à supporter les nouvelles modifications.

Pour s'assurer de la testabilité du logiciel, on pourrait proposer d'adopter une démarche TDD (*Test-Driven Development*). Cette démarche consiste à écrire les tests unitaires avant d'écrire le code pour la fonctionnalité. De ce fait, on s'assure que l'ensemble du programme est testable.

2. Fiabilité

Afin d'améliorer la tolérance aux pannes, qui est un critère essentiel à la fiabilité, on pourrait commencer par implémenter un mode dégradé de l'application. Ce mode permettrait de faire en sorte que si une fonctionnalité cause une panne, seule cette fonctionnalité devient inutilisable plutôt que l'ensemble du programme.

On remarque qu'il n'existe pas de tests d'intégration dans le code fourni par Weka. Les tests d'intégration servent à tester l'ensemble du système, cette étape se fait après les tests unitaires pour s'assurer qu'il n'y ait pas de problèmes dans l'interaction entre les modules. On propose alors une stratégie pour ce type de test: la stratégie "*Bottom-up*", qui part du module inférieur en montant jusqu'au module le plus supérieur on utilisant des pilotes.

Pour le deuxième sous-critère de fiabilité, on propose d'utiliser une stratégie "*fail-safe*" pour tester la recouvrabilité de l'application. Par exemple, on pourrait redémarrer la machine pendant que Weka roule, et vérifier si les données sont encore valides.

3. Aptitude fonctionnelle

Une façon de tester la complétude et l'exactitude fonctionnelle de Weka, serait de créer une matrice de respect des exigences. Dans cette matrice, on énumérerait l'ensemble des fonctionnalités et on vérifierait pour chacune si elle est bien implémentée et si elle respecte bien les exigences fonctionnelles.

C. RAPPORT DES TESTS

1. Maintenabilité

a) Testabilité

En lançant la totalité des tests avec "*JUnit*", on obtient les résultats suivants:

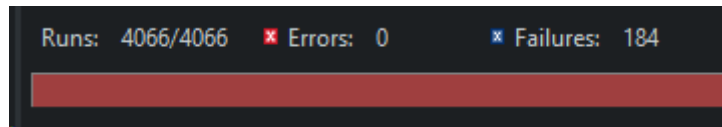


Figure 1: Résultat des tests unitaires obtenus avec JUnit

On observe que la majorité des tests passent et seulement 4% des tests échouent. Si on regarde cela plus en détails, on peut voir que tous les cas de test qui ne passent pas sont des tests de régression.

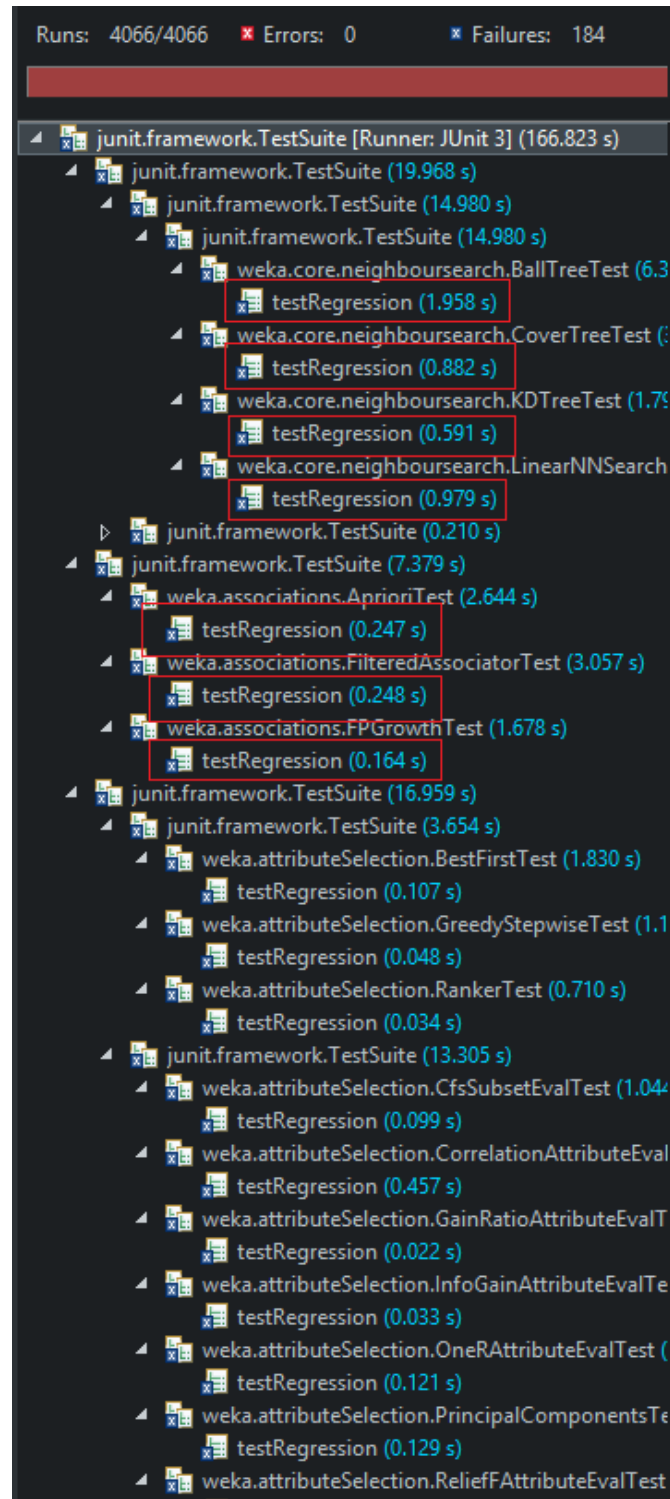


Figure 2: Liste des cas de tests qui échouent

Nous pensons que la raison pour laquelle ces tests échouent est que étant donné que ce sont des tests de régression, ils doivent posséder une version de base sur laquelle se référer. Comme nous n'en fournissons pas, cela pourrait expliquer pourquoi ceux-ci échouent.

Pour ce qui est de la couverture du code, nous avons généré un rapport de couverture grâce à l'outil "Jacoco". Dans l'ensemble on obtient la couverture suivante pour la totalité du système Weka:



Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
weka		31%		23%	54,212	68,758	153,379	210,235	20,767	28,319	2,414	3,427
Total	686,325 of 1,003,620	31%	60,902 of 80,086	23%	54,212	68,758	153,379	210,235	20,767	28,319	2,414	3,427

Figure 3: Couverture du code totale de l'ensemble de Weka

On observe une couverture de code de 31%, ce qui correspond à un pourcentage inférieur à la valeur attendue dans l'objectif de testabilité qui est 70%. Il faut donc ajouter d'autres tests unitaires afin de viser une meilleure couverture.

Nous avons ensuite isolé cinq fonctionnalités que nous avons étudié un peu plus en détails.

Classifieurs:





NaiveBayes		24%		26%	98	131	291	433	15	29	0	1
J48		49%		38%	90	104	136	257	53	65	0	1

Figure 4: Couverture du code dans les algorithmes de classification

Nous avons choisi deux algorithmes de classification: "Naïve Bayes" et "J48". Pour le premier, on observe une couverture de code de 24% et une couverture de branche de 26%, ce qui est en dessous de la moyenne pour le système weka.

Pour "J48", sa couverture de code est de 49% et sa couverture d'instruction est de 38%. Même s'il y a une différence significative entre les deux métriques, leurs valeurs sont quand même au-dessus de la moyenne globale (31%).

Clusters:





SimpleKMeans		58%		55%	142	256	356	881	34	74	0	1
EM		73%		68%	87	220	171	715	21	68	0	1

Figure 5: Couverture du code dans les algorithmes de clustering

Pour les algorithmes de clustering on peut voir que les valeurs sont relativement élevées ce qui témoigne de l'application de bonnes pratiques.

Association:

Apriori		54%		39%	153	194	347	679	51	67	0	1
---------	---	-----	---	-----	-----	-----	-----	-----	----	----	---	---

Figure 6: Couverture du code dans l'algorithme d'association

Même chose pour l'algorithme d'association qui possède 54% de couverture de code.

Vous pouvez consulter les rapports de couvertures détaillés pour ces fonctionnalités en annexe.

Afin d'augmenter la couverture du code, on pourrait instaurer des standards pour les nouveaux tests. Par exemple, on pourrait requérir aux développeurs de respecter la stratégie de couverture des instructions, c'est-à-dire d'essayer de couvrir toutes les lignes de code dans les tests unitaires.

b) Modifiabilité

Le code source de Weka possède plusieurs tests de régression que l'on retrouve pour chaque fonctionnalité qui a été développée. Un test de régression permet de lancer une série de tests et vérifier si les résultats obtenus sont les mêmes que ceux d'une version de référence. Ainsi, cela permet de vérifier qu'une même fonctionnalité fonctionne toujours correctement après une mise à jour du logiciel.

En lançant les tests de Weka, on obtient le rapport suivant:

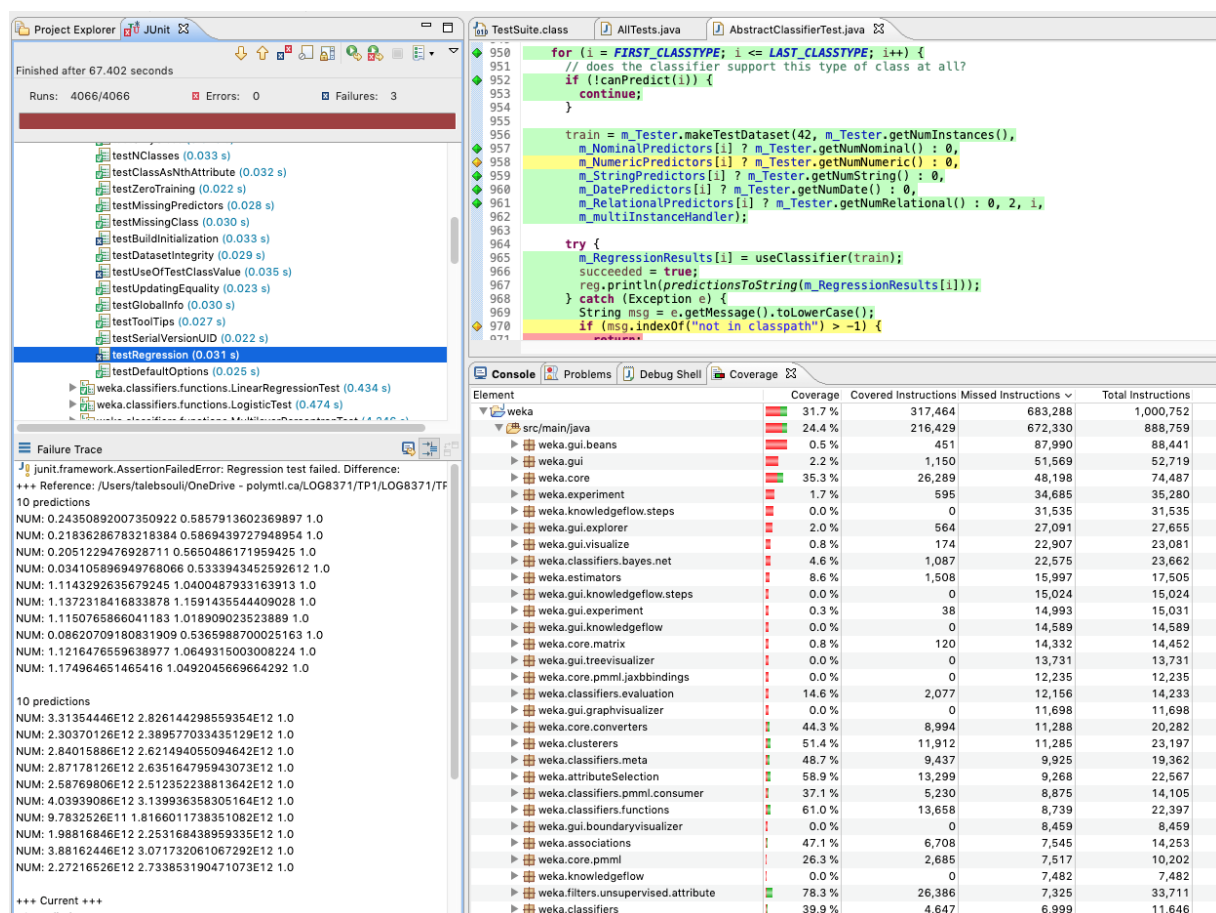


Figure 7: Couverture qui montre le bon fonctionnement des tests de régression

Dans celui-ci, on observe que tous les tests de régression passent, à l'exception de la classe "GaussianProcesses". Ceci montre donc qu'il y a un problème qui s'est

introduit dans Weka. Cependant, comme il ne s'agit pas d'un problème qui causerait une défaillance système, on peut considérer l'objectif comme atteint.

Afin de renforcer le critère de modifiabilité de Weka, nous pouvons configurer un outil d'intégration continue comme Jenkins, qui exécuterait tous les tests de régression et nous permettrait de plus facilement tracer quand est-ce qu'un problème de régression s'est introduit.

2. Fiabilité

a) Tolérance aux fautes

Pour la tolérance aux fautes, il faudrait tenter de tester si notre système est résilient en cas de fautes inattendues, comme une absence de données. Afin de tester ce type de situations, Weka possède des cas de tests pour les fonctionnalités critiques, qui permettent de vérifier qu'un algorithme fonctionne toujours bien en cas de fautes. Parmi ces cas de tests, on retrouve "*testMissingPredictors*" qui teste si le système fonctionne bien en cas d'absence de prédicteur ou encore "*testZeroTraining*" qui teste si l'algorithme donne des résultats cohérents en cas d'absence d'entraînement.

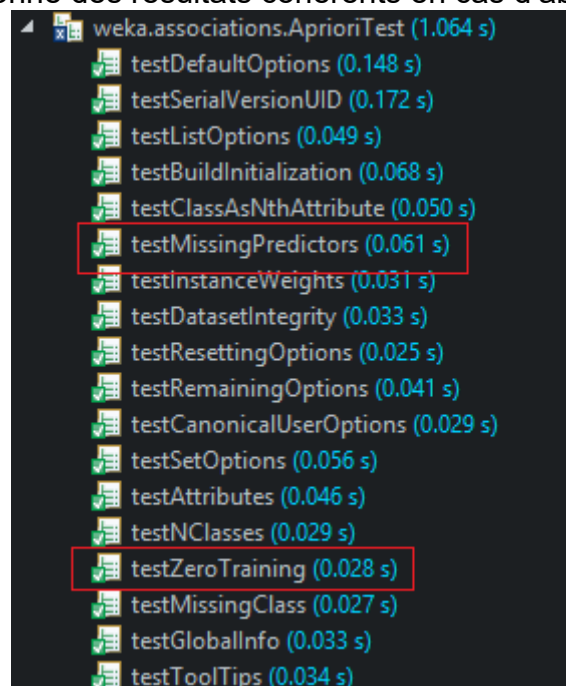


Figure 8: Exemple de Test *testMissingPredictors* et *testZeroTraining*

Dans la capture d'écran ci-dessus, on voit un exemple de ces cas de tests pour l'algorithme d'association à priori.

Comme nous l'avons vu dans les parties précédentes du rapport de tests, ces tests passent pour toutes les classes de Weka, ce qui nous montre que l'objectif visé de tolérance aux fautes est atteint.

b) Récupérabilité

Au moment de l'exécution de Weka, si le logiciel s'arrête instantanément, on doit être capable de le rouvrir et de repartir de là où on était. Ainsi, le logiciel doit nous proposer de revenir à la version juste avant l'arrêt, avec les dernières données et la dernière configuration mise en place.

La figure suivante montre que l'on a exécuté Weka et que l'on a ouvert un fichier "*.arff" sur lequel on va travailler.

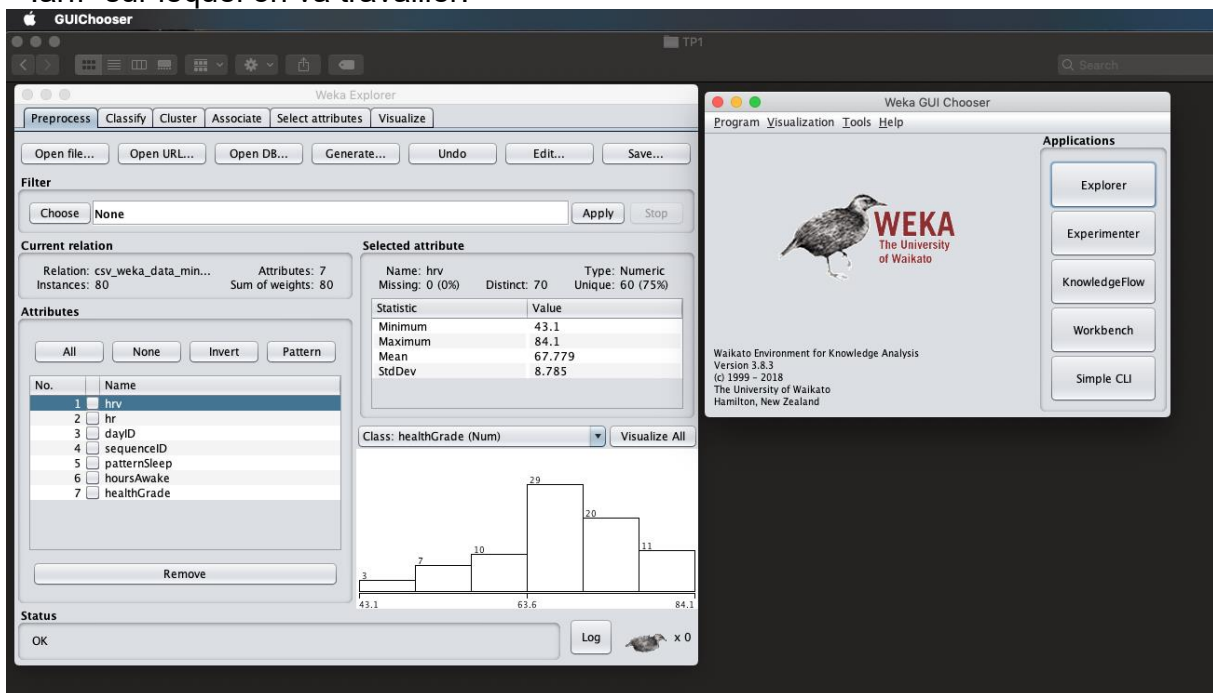


Figure 9: Exécution de Weka et ouverture d'un fichier *.arff

La figure suivante montre que l'on a apporté des nouveaux changements aux données du fichier "*.arff" et à la configuration des attributs sur Weka.

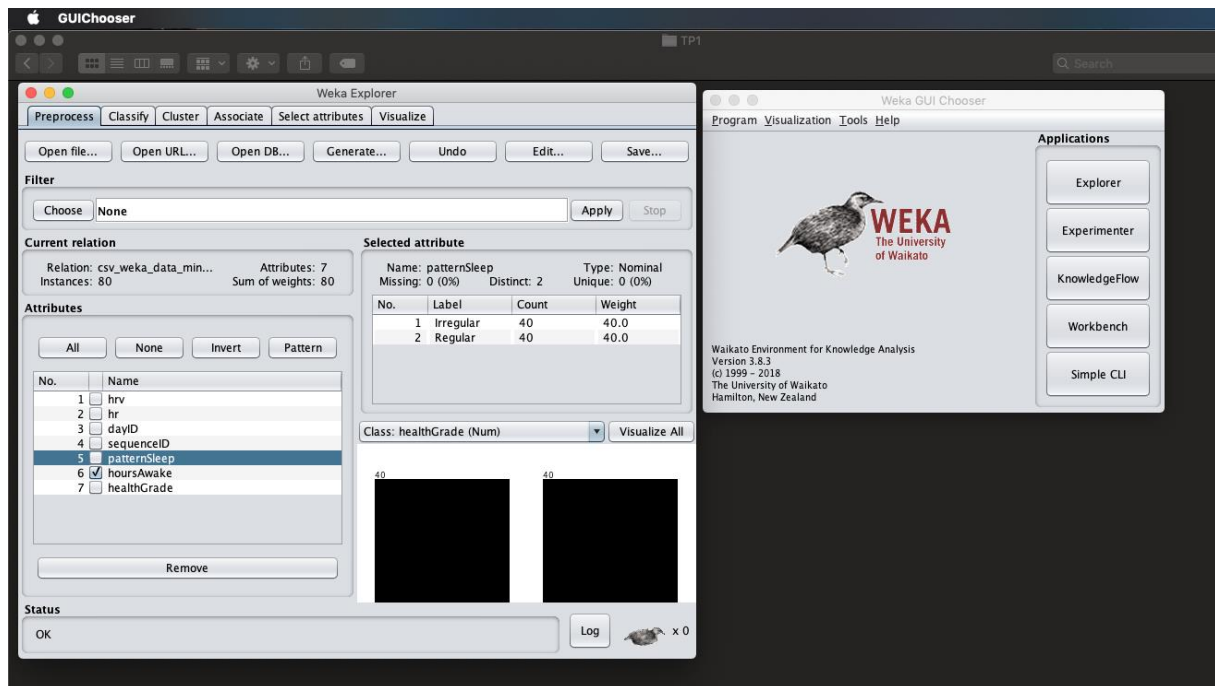


Figure 10: Changements apportés aux données

La figure suivante montre l'opération de fermeture instantanée de Weka, afin de le rouvrir et de vérifier si la configuration et les données sont toujours sauvegardés et valides .

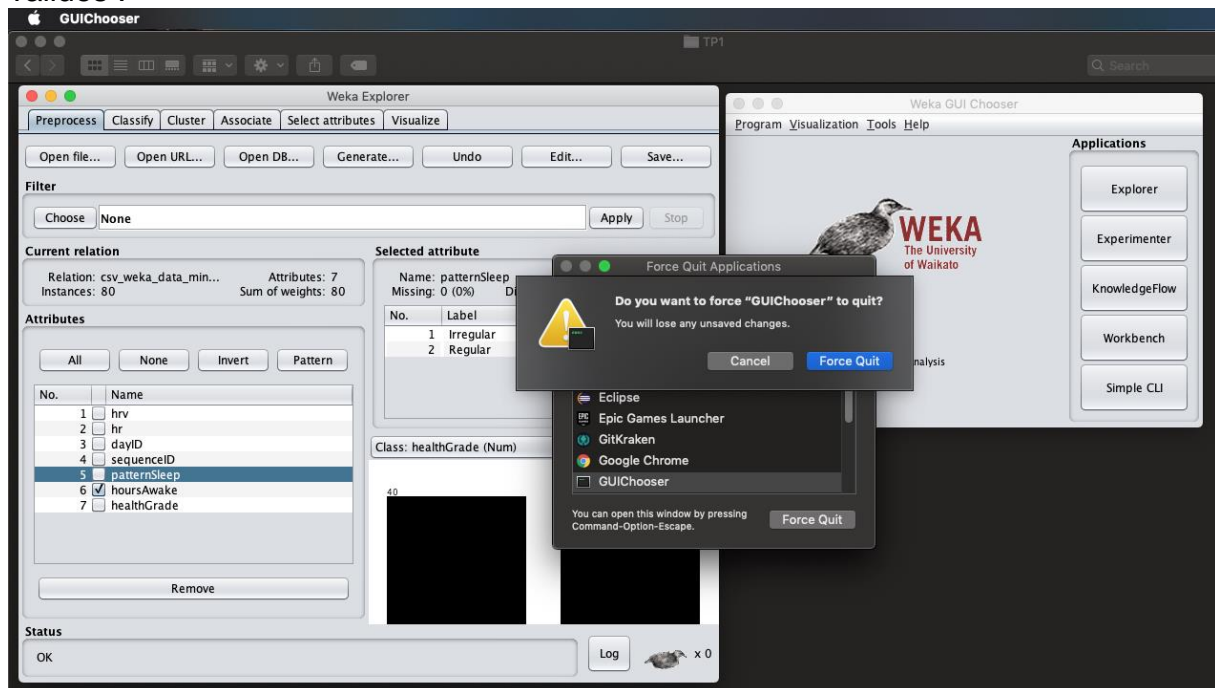


Figure 11: Forcer la fermeture de Weka

La figure suivante montre que l'on a ré-exécuté Weka et que l'on a ouvert un fichier "*.arff" sur lequel on a déjà travaillé, on remarque que la configuration des attributs et les données dernièrement utilisées ne sont pas maintenues par le logiciel. On revient à la première version avant les changements.

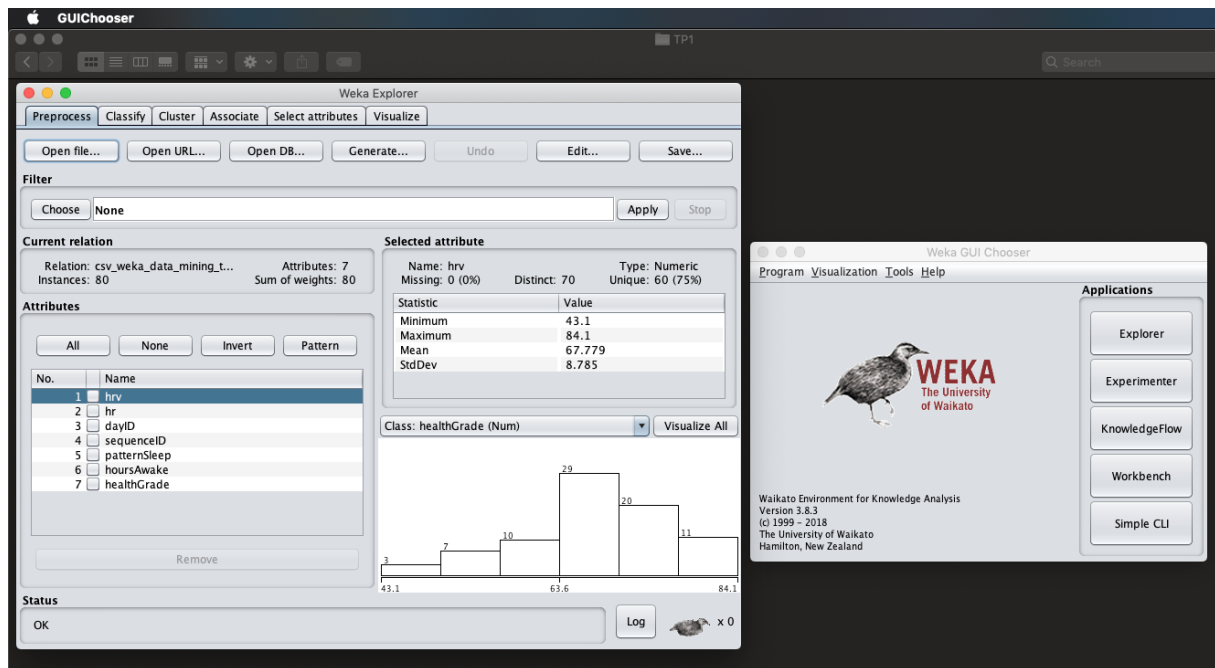


Figure 12: Réexécution de Weka

3. Aptitude fonctionnelle

a) Complétude fonctionnelle

Afin de tester la complétude fonctionnelle du logiciel Weka, nous allons intégrer un outil d'automatisation des tests fonctionnelles: “*Squash TA*”. Ce type d'outil permet de réaliser des tests en fonction des exigences fonctionnelles de notre logiciel et de les automatiser.

On peut donc créer des suites de tests qui reflètent les exigences fonctionnelles identifiées dans le SRS de l'application, et de vérifier si chacune de ces exigences est bien respectée.

Pour valider l'objectif de complétude fonctionnelle que nous avons établi, il faudrait donc exécuter les tests fonctionnels développés sur “*Squash TA*” et vérifier que le SRS est bien respecté.

b) Exactitude fonctionnelle

Ce sous critère de l'aptitude fonctionnelle nous permet de vérifier la précision des résultats fournis par Weka. Afin de vérifier l'exactitude, on doit exécuter les tests unitaires conçus par Weka et vérifier que les résultats obtenus soient les mêmes que ceux qui sont attendus. Cette vérification est effectuée dans les tests unitaires qui prennent des valeurs en entrée et comparent les résultats obtenus par l'exécution d'un algorithme avec les résultats qui sont attendus.

On peut donc exécuter les tests de Weka et analyser les résultats de ce type de tests, qui correspond au total des tests unitaire déjà présents auquel on soustrait le

nombre de tests de régression. Sur ceux-ci, deux échouent. Ceci témoigne donc d'une bonne exactitude fonctionnelle étant donné qu'environ 99.5% des classes possèdent le comportement attendu.

III. Question 3: Plan d'intégration continue

Pour ce qui trait à l'intégration continue, nous avons décidé d'utiliser Jenkins. Cet outil nous permet d'automatiser l'exécution des tests (unitaires et d'intégration) et de faire en sorte qu'il s'exécute à chaque modification apporté au logiciel.

Pour exécuter Jenkins, nous le lançons en local sur un serveur Jetty avec la commande suivante:

```
admin@dell MINGW64 /c/Program Files (x86)/Jenkins
$ java -jar jenkins.war
```

Figure 13: Lancement de Jenkins

Une fois la commande exécutée et Jenkins lancé, nous avons créé un "job" Jenkins qui clone le projet Weka et exécute ensuite le build ant permettant de compiler et lancer l'ensemble des tests contenus dans le test suite de Weka.



Figure 14: Configuration de "Ant"

Ainsi, à chaque fois que ce build est exécuté, les tests le sont aussi. Ensuite, on peut faire en sorte que le build soit déclenché à chaque fois qu'un changement a été poussé sur le git de weka en cochant l'option suivante:

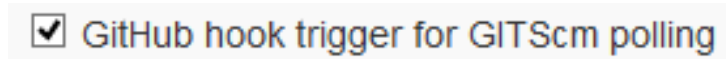


Figure 15: Choix de déclencheur du build

Il faut ensuite configurer le projet github en ajoutant un "webhook" vers notre instance jenkins pour qu'il envoie une notification à chaque fois qu'un "push" est fait.

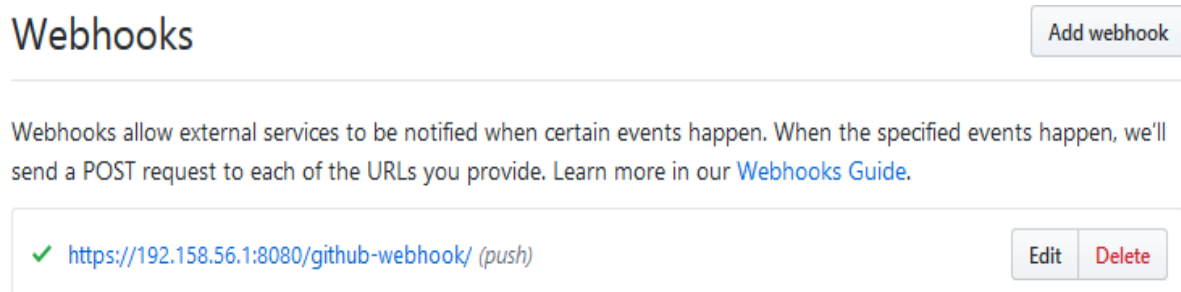


Figure 16: Configurer le projet github

En exécutant le “*job*” jenkins sur le projet weka, on obtient les résultats suivants:

```
[junit] FAILURES!!!  
[junit] Tests run: 4065, Failures: 183, Errors: 0  
[junit]  
  
BUILD FAILED  
C:\Users\admin\Documents\Poly\LOG8371\LOG8371\TP1\weka\build.xml:778: Java returned: 1  
  
Total time: 1 minute 23 seconds  
Build step 'Invoke Ant' marked build as failure  
Recording test results  
Finished: FAILURE
```

Figure 17: Résultat du job jenkins

On observe bien que tous les tests sont lancés, mais que le build échoue. La raison de cet échec est que les tests de régression, comme nous l'avons vu plus tôt, ne passent pas, ce qui explique aussi les 183 échecs. Les résultats obtenus dans le build sont les mêmes que ceux obtenus en lançant les tests sur JUnit, ce qui montre bien que le “*job*” que nous avons créé fonctionne.

IV. Question 4

A. « NOUVEAU » ALGORITHME POUR WEKA

Nous avons ajouté une nouvelle fonctionnalité à Weka dans un nouveau package *weka.newFunction*. Cette fonctionnalité ne fait que retourner un string *“Hello World”* et a été ajoutée afin de montrer comment l’ajout d’une nouvelle fonctionnalité impacte le plan d’assurance qualité.

```
1 package weka.newFunction;
2
3 public class NewFunction {
4     public String helloWorld() {
5         return "Hello World";
6     }
7 }
8
```

Figure 18: Nouvelle fonctionnalité « NewFunction »

B. GARANTIR LA QUALITE DU SYSTEME APRES LES MODIFICATIONS

Afin de s’assurer de la qualité du système, nous avons ajouté un test unitaire, pour la fonction nouvellement ajoutée dans Weka.

```
package weka.newFunction;

import junit.framework.Test;

public class NewFunctionTest extends TestCase {

    public NewFunctionTest(String name) {
        super(name);
    }

    public void testHelloWorld() {
        NewFunction f = new NewFunction();
        junit.framework.Assert.assertEquals(f.helloWorld(), "Hello World");
    }

    public static Test suite() {
        return new TestSuite(NewFunctionTest.class);
    }

    public static void main(String []args) {
        junit.textui.TestRunner.run(suite());
    }
}
```

Figure 19: Test unitaire ajouté pour tester la nouvelle fonctionnalité

Nous l’avons aussi ajouté au test suite qui contient l’ensemble des autres tests unitaires.

```
// New Function|
suite.addTest(weka.newFunction.AllTests.suite());
```

Figure 20: Ligne de code ajouté dans le fichier test « weka.AllTests »

Ainsi, ce test sera automatiquement lancé avec tout le reste des tests unitaires.

C. LES TESTS NECESSAIRES

En plus des tests unitaires que nous avons ajoutés, il faudrait aussi tester l'intégration de cette nouvelle fonctionnalité avec le reste du système Weka grâce à des tests d'intégration comme nous l'avons mentionné précédemment. En revanche, comme dans notre exemple, il s'agit d'une fonctionnalité très basique, ajouter des tests d'intégration pour celle-ci est beaucoup moins pertinent étant donné qu'elle n'interagit avec rien.

D. LA MISE A JOUR DU PLAN DE QUALITE

Etant donné que le plan de qualité s'intéresse à la qualité globale du logiciel et non à une fonctionnalité en particulier, il s'adapte facilement à l'ajout d'une fonctionnalité. En effet les règles d'assurance qualité que nous avons spécifiée dans les parties précédentes s'appliquent aussi. Comme il ne s'agit pas d'une nouvelle fonctionnalité majeure, comme le passage à une plateforme mobile, qui demanderait des modifications significatives sur le plan de test, nous n'avons pas à mettre à jour le plan de qualité. Les objectifs de qualité restent les mêmes qu'avant et il suffit d'ajouter les nouveaux tests dans le test suite de Weka et qui seront ensuite lancés avec les autres. C'est l'un des avantages de l'intégration continue.

Révision	Date	Auteur	Commentaires
	18/01/2019	Equipe Romeo	Première ébauche
1.0	01/02/2019	Equipe Romeo	Première version du plan de qualité.
1.1	09/02/2019	Equipe Romeo	Modification du plan de qualité.
2.0	21/02/2019	Equipe Romeo	Version finale du plan de qualité.

Tableau 8: Détails des révisions

V.Question 5 : Vidéo

Nous joindrons une vidéo afin de démontrer le bon fonctionnement de l'intégration continue : <https://youtu.be/0CAV77w0FJI>

























Références

- [1] R. R. Bouckaert, E. Frank, M. Hall, R. Kirkby, P. Reutemann, A. Seewald, D. Scuse. *WEKA Manual* Version 3.7.8.(2013). Consulté le: Janvier. 16-27, 2019. [en ligne]. disponible:
http://statweb.stanford.edu/~lpekelis/13_datafest_cart/WekaManual-3-7-8.pdf
- [2] D. GALIN, *Software Quality Assurance From theory to implementation*, vol. 1. Pearson Education Limited, Edinburgh Gate, Harlow, Essex CM20 2JE, England, 2004, pp. 48–50. Consulté le: Janvier. 16-27, 2019. [en ligne]. disponible:
<http://desy.lecturer.pens.ac.id/Manajemen%20Kualitas%20Perangkat%20Lunak/ebook/Software%20Quality%20Assurance%20From%20Theory%20to%20Implementation.pdf>
- [3] C. Y. Laporte, A. April, *Software Quality Assurance*, vol. 1. John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, USA, 2018, pp. 76, 81,82,. Consulté le: Janvier. 16-27, 2019. [en ligne]. disponible:
http://olea.org/recursos/2017-Laporte_April-Software_Quality_Assurance/2017-Laporte,%20April-Software%20Quality%20Assurance,%20First%20Edition-9781119312451.pdf
- [4] Systems and software engineering — Systems and software product Quality Requirements and Evaluation (SQuaRE) — Quality measure elements, 3rd ed., DRAFT INTERNATIONAL STANDARD ISO/IEC DIS 25021,2011, pp. 18,19,32-35.

Annexe

Les figures ci-dessous montrent en détails la couverture de code de chaque algorithme choisi.

NaiveBayes

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
● aggregate(NaiveBayes)		0%		0%	6	6	12	12	1	1
● buildClassifier(Instances)		85%		85%	4	15	7	55	0	1
● displayModelInOldFormatTipText()		0%		n/a	1	1	1	1	1	1
● distributionForInstance(Instance)		79%		85%	3	11	5	29	0	1
● finalizeAggregation()		0%		n/a	1	1	1	1	1	1
● getCapabilities()		100%		n/a	0	1	0	9	0	1
● getClassEstimator()		0%		n/a	1	1	1	1	1	1
● getConditionalEstimators()		100%		n/a	0	1	0	1	0	1
● getDisplayModelInOldFormat()		0%		n/a	1	1	1	1	1	1
● getHeader()		100%		n/a	0	1	0	1	0	1
● getOptions()		66%		50%	3	4	3	9	0	1
● getRevision()		0%		n/a	1	1	1	1	1	1
● getTechnicalInformation()		0%		n/a	1	1	11	11	1	1
● getUseKernelEstimator()		0%		n/a	1	1	1	1	1	1
● getUseSupervisedDiscretization()		0%		n/a	1	1	1	1	1	1
● globalInfo()		0%		n/a	1	1	2	2	1	1
● listOptions()		100%		n/a	0	1	0	13	0	1
● main(String[])		0%		n/a	1	1	2	2	1	1
● NaiveBayes()		100%		n/a	0	1	0	5	0	1
● pad(String, String, int, boolean)		0%		0%	4	4	10	10	1	1
● setDisplayModelInOldFormat(boolean)		100%		n/a	0	1	0	2	0	1
● setOptions(String[])		79%		25%	2	3	1	10	0	1
● setUseKernelEstimator(boolean)		66%		50%	1	2	1	4	0	1
● setUseSupervisedDiscretization(boolean)		66%		50%	1	2	1	4	0	1
● toString()		1%		1%	56	57	206	213	0	1
● toStringOriginal()		0%		0%	5	5	21	21	1	1
● updateClassifier(Instance)		100%		83%	1	4	0	11	0	1
● useKernelEstimatorTipText()		0%		n/a	1	1	1	1	1	1
● useSupervisedDiscretizationTipText()		0%		n/a	1	1	1	1	1	1
Total	1,673 of 2,211	24%	150 of 203	26%	98	131	291	433	15	29

Annexe_Figure 1: Couvrage de l'algorithme « NaiveBayes »

J48

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
binarySplitsTipText()		0%		n/a	1	1	1	1	1	1
buildClassifier(Instances)		55%		35%	7	8	7	19	0	1
classifyInstance(Instance)		100%		n/a	0	1	0	1	0	1
collapseTreeTipText()		0%		n/a	1	1	1	1	1	1
confidenceFactorTipText()		0%		n/a	1	1	1	1	1	1
distributionForInstance(Instance)		100%		n/a	0	1	0	1	0	1
doNotMakeSplitPointActualValueTipText()		0%		n/a	1	1	1	1	1	1
enumerateMeasures()		0%		n/a	1	1	5	5	1	1
generatePartition(Instances)		100%		n/a	0	1	0	2	0	1
getBinarySplits()		0%		n/a	1	1	1	1	1	1
getCapabilities()		100%		n/a	0	1	0	10	0	1
getCollapseTree()		0%		n/a	1	1	1	1	1	1
getConfidenceFactor()		0%		n/a	1	1	1	1	1	1
getDoNotMakeSplitPointActualValue()		0%		n/a	1	1	1	1	1	1
getMeasure(String)		0%		0%	4	4	8	8	1	1
getMembershipValues(Instance)		100%		n/a	0	1	0	1	0	1
getMinNumObj()		0%		n/a	1	1	1	1	1	1
getNumFolds()		0%		n/a	1	1	1	1	1	1
getOptions()		51%		50%	9	10	15	31	0	1
getReducedErrorPruning()		0%		n/a	1	1	1	1	1	1
getRevision()		0%		n/a	1	1	1	1	1	1
getSaveInstanceData()		0%		n/a	1	1	1	1	1	1
getSeed()		0%		n/a	1	1	1	1	1	1
getSubtreeRaising()		0%		n/a	1	1	1	1	1	1
getTechnicalInformation()		0%		n/a	1	1	7	7	1	1
getUnpruned()		0%		n/a	1	1	1	1	1	1
getUseLaplace()		0%		n/a	1	1	1	1	1	1
getUseMDLcorrection()		0%		n/a	1	1	1	1	1	1
globalInfo()		0%		n/a	1	1	2	2	1	1
graph()		0%		n/a	1	1	1	1	1	1
graphType()		0%		n/a	1	1	1	1	1	1
J48()		100%		n/a	0	1	0	13	0	1
listOptions()		100%		n/a	0	1	0	29	0	1
main(String[])		0%		n/a	1	1	2	2	1	1
measureNumLeaves()		0%		n/a	1	1	1	1	1	1
measureNumRules()		0%		n/a	1	1	1	1	1	1
measureTreeSize()		0%		n/a	1	1	1	1	1	1
minNumObjTipText()		0%		n/a	1	1	1	1	1	1
numElements()		100%		n/a	0	1	0	1	0	1
numFoldsTipText()		0%		n/a	1	1	1	1	1	1
prefix()		0%		n/a	1	1	1	1	1	1
reducedErrorPruningTipText()		0%		n/a	1	1	1	1	1	1
saveInstanceDataTipText()		0%		n/a	1	1	1	1	1	1
seedTipText()		0%		n/a	1	1	1	1	1	1
setBinarySplits(boolean)		0%		n/a	1	1	2	2	1	1
setCollapseTree(boolean)		0%		n/a	1	1	2	2	1	1
setConfidenceFactor(float)		0%		n/a	1	1	2	2	1	1
setDoNotMakeSplitPointActualValue(boolean)		0%		n/a	1	1	2	2	1	1
setMinNumObj(int)		0%		n/a	1	1	2	2	1	1
setNumFolds(int)		0%		n/a	1	1	2	2	1	1
setOptions(String[])		70%		46%	14	17	15	48	0	1
setReducedErrorPruning(boolean)		0%		0%	2	2	4	4	1	1
setSaveInstanceData(boolean)		0%		n/a	1	1	2	2	1	1
setSeed(int)		0%		n/a	1	1	2	2	1	1
setSubtreeRaising(boolean)		0%		n/a	1	1	2	2	1	1
setUnpruned(boolean)		0%		0%	2	2	4	4	1	1
setUseLaplace(boolean)		0%		n/a	1	1	2	2	1	1
setUseMDLcorrection(boolean)		0%		n/a	1	1	2	2	1	1
subtreeRaisingTipText()		0%		n/a	1	1	1	1	1	1
toSource(String)		0%		n/a	1	1	7	7	1	1
toString()		17%		25%	2	3	3	5	0	1
toSummaryString()		0%		n/a	1	1	2	2	1	1
unprunedTipText()		0%		n/a	1	1	1	1	1	1
useLaplaceTipText()		0%		n/a	1	1	1	1	1	1
useMDLcorrectionTipText()		0%		n/a	1	1	1	1	1	1
Total	489 of 973	49%	48 of 78	38%	90	104	136	257	53	65

Annexe_Figure 2: Couverture de l'algorithme « J48 »

SimpleKMeans

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
● toString()		50%		55%	35	57	120	250	0	1
● buildClusterer(Instances)		69%		67%	23	47	48	165	0	1
● kMeansPlusPlusInit(Instances)		0%		0%	12	12	51	51	1	1
● moveCentroid(int, Instances, boolean, boolean)		68%		60%	11	24	19	57	0	1
● launchAssignToClusters(Instances, int[])		0%		0%	5	5	16	16	1	1
● launchMoveCentroids(Instances[])		0%		0%	4	4	14	14	1	1
● canopyInit(Instances)		0%		0%	2	2	14	14	1	1
● clusterProcessedInstance(Instance, boolean, boolean, long[])		70%		55%	5	10	10	29	0	1
● getTechnicalInformation()		0%		n/a	1	1	8	8	1	1
● getOptions()		90%		50%	5	6	5	34	0	1
● getAssignments()		0%		0%	3	3	6	6	1	1
● farthestFirstInit(Instances)		0%		n/a	1	1	5	5	1	1
● globalInfo()		0%		n/a	1	1	2	2	1	1
● setDistanceFunction(DistanceFunction)		46%		25%	2	3	3	6	0	1
● main(String[])		0%		n/a	1	1	2	2	1	1
● setOptions(String[])		97%		95%	1	12	1	47	0	1
● setNumClusters(int)		54%		50%	1	2	1	4	0	1
● setMaxIterations(int)		54%		50%	1	2	1	4	0	1
● setReduceNumberOfDistanceCalcsViaCanopies(boolean)		0%		n/a	1	1	2	2	1	1
● setPreserveInstancesOrder(boolean)		0%		n/a	1	1	2	2	1	1
● setFastDistanceCalc(boolean)		0%		n/a	1	1	2	2	1	1
● getReduceNumberOfDistanceCalcsViaCanopies()		0%		n/a	1	1	1	1	1	1
● getDisplayStdDevs()		0%		n/a	1	1	1	1	1	1
● getDontReplaceMissingValues()		0%		n/a	1	1	1	1	1	1
● getDistanceFunction()		0%		n/a	1	1	1	1	1	1
● getPreserveInstancesOrder()		0%		n/a	1	1	1	1	1	1
● getFastDistanceCalc()		0%		n/a	1	1	1	1	1	1
● getRevision()		0%		n/a	1	1	1	1	1	1
● clusterInstance(Instance)		92%		50%	1	2	1	8	0	1
● getSquaredError()		77%		50%	1	2	1	3	0	1
● numClustersTipText()		0%		n/a	1	1	1	1	1	1
● initializationMethodTipText()		0%		n/a	1	1	1	1	1	1
● reduceNumberOfDistanceCalcsViaCanopiesTipText()		0%		n/a	1	1	1	1	1	1
● canopyPeriodicPruningRateTipText()		0%		n/a	1	1	1	1	1	1
● canopyMinimumCanopyDensityTipText()		0%		n/a	1	1	1	1	1	1
● canopyMaxNumCanopiesToHoldInMemoryTipText()		0%		n/a	1	1	1	1	1	1
● canopyT2TipText()		0%		n/a	1	1	1	1	1	1
● canopyT1TipText()		0%		n/a	1	1	1	1	1	1
● maxIterationsTipText()		0%		n/a	1	1	1	1	1	1
● displayStdDevsTipText()		0%		n/a	1	1	1	1	1	1
● dontReplaceMissingValuesTipText()		0%		n/a	1	1	1	1	1	1
● distanceFunctionTipText()		0%		n/a	1	1	1	1	1	1
● preserveInstancesOrderTipText()		0%		n/a	1	1	1	1	1	1
● fastDistanceCalcTipText()		0%		n/a	1	1	1	1	1	1
● numExecutionSlotsTipText()		0%		n/a	1	1	1	1	1	1
● listOptions()		100%		n/a	0	1	0	51	0	1
● SimpleKMeans()		100%		n/a	0	1	0	20	0	1
● pad(String, String, int, boolean)		100%		100%	0	4	0	10	0	1
● static { ... }		100%		n/a	0	1	0	3	0	1
● getCapabilities()		100%		n/a	0	1	0	7	0	1
● startExecutorPool()		100%		100%	0	2	0	4	0	1
● setInitializationMethod(SelectedTag)		100%		50%	1	2	0	3	0	1
● getInitializationMethod()		100%		n/a	0	1	0	1	0	1
● setCanopyPeriodicPruningRate(int)		100%		n/a	0	1	0	2	0	1
● setCanopyMinimumCanopyDensity(double)		100%		n/a	0	1	0	2	0	1
● setCanopyMaxNumCanopiesToHoldInMemory(int)		100%		n/a	0	1	0	2	0	1
● setCanopyT2(double)		100%		n/a	0	1	0	2	0	1
● setCanopyT1(double)		100%		n/a	0	1	0	2	0	1
● setDisplayStdDevs(boolean)		100%		n/a	0	1	0	2	0	1
● setDontReplaceMissingValues(boolean)		100%		n/a	0	1	0	2	0	1
● setNumExecutionSlots(int)		100%		n/a	0	1	0	2	0	1
● numberOfClusters()		100%		n/a	0	1	0	1	0	1
● getNumClusters()		100%		n/a	0	1	0	1	0	1
● getCanopyPeriodicPruningRate()		100%		n/a	0	1	0	1	0	1
● getCanopyMinimumCanopyDensity()		100%		n/a	0	1	0	1	0	1
● getCanopyMaxNumCanopiesToHoldInMemory()		100%		n/a	0	1	0	1	0	1
● getCanopyT2()		100%		n/a	0	1	0	1	0	1
● getCanopyT1()		100%		n/a	0	1	0	1	0	1
● getMaxIterations()		100%		n/a	0	1	0	1	0	1
● getNumExecutionSlots()		100%		n/a	0	1	0	1	0	1
● getClusterCentroids()		100%		n/a	0	1	0	1	0	1
● getClusterStandardDevs()		100%		n/a	0	1	0	1	0	1
● getClusterNominalCounts()		100%		n/a	0	1	0	1	0	1
● getClusterSizes()		100%		n/a	0	1	0	1	0	1
Total	1,655 of 3,977	58%	161 of 362	55%	142	256	356	881	34	74








































Annexe_Figure 3: Couverture de l'algorithme « SimpleKMeans »

EM

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
• buildClusterer(Instances)		100%		100%	0	3	0	20	0	1
• clusterPriors()		100%		n/a	0	1	0	3	0	1
• CVClusters()		67%		57%	10	15	21	66	0	1
• debugTipText()		0%		n/a	1	1	1	1	1	1
• displayModelInOldFormatTipText()		0%		n/a	1	1	1	1	1	1
• doEM()		71%		71%	4	8	4	25	0	1
• E(Instances, boolean)		100%		100%	0	4	0	10	0	1
• EM()		100%		n/a	0	1	0	12	0	1
• EM_Init(Instances)		97%		88%	3	14	1	53	0	1
• EM_Report(Instances)		0%		0%	7	7	22	22	1	1
• estimate_priors(Instances)		100%		100%	0	4	0	8	0	1
• getCapabilities()		100%		n/a	0	1	0	3	0	1
• getClusterModelsNumericAtts()		0%		n/a	1	1	1	1	1	1
• getClusterPriors()		0%		n/a	1	1	1	1	1	1
• getDebug()		100%		n/a	0	1	0	1	0	1
• getDisplayModelInOldFormat()		0%		n/a	1	1	1	1	1	1
• getMaximumNumberOfClusters()		100%		n/a	0	1	0	1	0	1
• getMaxIterations()		0%		n/a	1	1	1	1	1	1
• getMinLogLikelihoodImprovementCV()		100%		n/a	0	1	0	1	0	1
• getMinLogLikelihoodImprovementIterating()		100%		n/a	0	1	0	1	0	1
• getMinStdDev()		100%		n/a	0	1	0	1	0	1
• getNumClusters()		100%		n/a	0	1	0	1	0	1
• getNumExecutionSlots()		100%		n/a	0	1	0	1	0	1
• getNumFolds()		100%		n/a	0	1	0	1	0	1
• getNumKMeansRuns()		100%		n/a	0	1	0	1	0	1
• getOptions()		97%		50%	1	2	1	23	0	1
• getRevision()		0%		n/a	1	1	1	1	1	1
• globalInfo()		0%		n/a	1	1	1	1	1	1
• iterate(Instances, boolean)		57%		66%	6	13	16	46	0	1
• launchESteps(Instances)		17%		10%	5	6	15	20	0	1
• launchMSteps(Instances)		2%		4%	12	13	34	37	0	1
• listOptions()		100%		n/a	0	1	0	36	0	1
• logDensityPerClusterForInstance(Instance)		100%		100%	0	5	0	14	0	1
• logNormalDens(double, double, double)		100%		n/a	0	1	0	3	0	1
• M(Instances)		100%		100%	0	5	0	16	0	1
• M_reEstimate(Instances)		82%		80%	4	11	5	25	0	1
• main(String[])		0%		n/a	1	1	2	2	1	1
• maximumNumberOfClustersTipText()		0%		n/a	1	1	1	1	1	1
• maxIterationsTipText()		0%		n/a	1	1	1	1	1	1
• minLogLikelihoodImprovementCVTipText()		0%		n/a	1	1	1	1	1	1
• minLogLikelihoodImprovementIteratingTipText()		0%		n/a	1	1	1	1	1	1
• minStdDevTipText()		0%		n/a	1	1	1	1	1	1
• new_estimators()		100%		100%	0	4	0	12	0	1
• numberOfClusters()		58%		50%	1	2	1	3	0	1
• numClustersTipText()		0%		n/a	1	1	1	1	1	1
• numExecutionSlotsTipText()		0%		n/a	1	1	1	1	1	1
• numFoldsTipText()		0%		n/a	1	1	1	1	1	1
• numKMeansRunsTipText()		0%		n/a	1	1	1	1	1	1
• pad(String, String, int, boolean)		100%		100%	0	4	0	10	0	1
• resetOptions()		100%		n/a	0	1	0	11	0	1
• setDebug(boolean)		100%		n/a	0	1	0	2	0	1
• setDisplayModelInOldFormat(boolean)		100%		n/a	0	1	0	2	0	1
• setMaximumNumberOfClusters(int)		100%		n/a	0	1	0	2	0	1
• setMaxIterations(int)		58%		50%	1	2	1	4	0	1
• setMinLogLikelihoodImprovementCV(double)		100%		n/a	0	1	0	2	0	1
• setMinLogLikelihoodImprovementIterating(double)		100%		n/a	0	1	0	2	0	1
• setMinStdDev(double)		100%		n/a	0	1	0	2	0	1
• setMinStdDevPerAtt(double[])		0%		n/a	1	1	2	2	1	1
• setNumClusters(int)		52%		50%	2	3	3	9	0	1
• setNumExecutionSlots(int)		100%		n/a	0	1	0	2	0	1
• setNumFolds(int)		100%		n/a	0	1	0	2	0	1
• setNumKMeansRuns(int)		100%		n/a	0	1	0	2	0	1
• setOptions(String[])		100%		100%	0	10	0	33	0	1
• startExecutorPool()		100%		100%	0	2	0	4	0	1
• static {...}		100%		n/a	0	1	0	1	0	1
• toString()		96%		90%	5	28	4	109	0	1
• toStringOriginal()		0%		0%	7	7	22	22	1	1
• updateMinMax(Instance)		100%		100%	0	4	0	7	0	1
Total	1,008 of 3,737	73%	97 of 304	68%	87	220	171	715	21	68

Annexe_Figure 4: Couverture de l'algorithme « EM »

Apriori

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
● Apriori()		100%		n/a	0	1	0	6	0	1
● buildAssociations(Instances)		86%		55%	21	31	20	107	0	1
● canProduceRules()		0%		n/a	1	1	1	1	1	1
● carTipText()		0%		n/a	1	1	1	1	1	1
● classIndexTipText()		0%		n/a	1	1	1	1	1	1
● deltaTipText()		0%		n/a	1	1	1	1	1	1
● findCarRulesQuickly()		0%		0%	4	4	13	13	1	1
● findLargeCartItemSets()		0%		0%	5	5	29	29	1	1
● findLargeItemSets()		89%		75%	2	5	4	29	0	1
● findRulesBruteForce()		0%		0%	4	4	18	18	1	1
● findRulesQuickly()		100%		87%	1	5	0	17	0	1
● getAllTheRules()		0%		n/a	1	1	1	1	1	1
● getAssociationRules()		0%		0%	11	11	39	39	1	1
● getCapabilities()		100%		n/a	0	1	0	8	0	1
● getCar()		0%		n/a	1	1	1	1	1	1
● getClassIndex()		0%		n/a	1	1	1	1	1	1
● getDelta()		0%		n/a	1	1	1	1	1	1
● getInstancesNoClass()		0%		n/a	1	1	1	1	1	1
● getInstancesOnlyClass()		0%		n/a	1	1	1	1	1	1
● getLowerBoundMinSupport()		0%		n/a	1	1	1	1	1	1
● getMetricType()		0%		n/a	1	1	1	1	1	1
● getMinMetric()		0%		n/a	1	1	1	1	1	1
● getNumRules()		0%		n/a	1	1	1	1	1	1
● getOptions()		81%		57%	6	8	7	34	0	1
● getOutputItemSets()		0%		n/a	1	1	1	1	1	1
● getRemoveAllMissingCols()		100%		n/a	0	1	0	1	0	1
● getRevision()		0%		n/a	1	1	1	1	1	1
● getRuleMetricNames()		0%		0%	2	2	4	4	1	1
● getSignificanceLevel()		0%		n/a	1	1	1	1	1	1
● getTechnicalInformation()		0%		n/a	1	1	19	19	1	1
● getTreatZeroAsMissing()		0%		n/a	1	1	1	1	1	1
● getUpperBoundMinSupport()		0%		n/a	1	1	1	1	1	1
● getVerbose()		0%		n/a	1	1	1	1	1	1
● globalInfo()		0%		n/a	1	1	2	2	1	1
● listOptions()		100%		n/a	0	1	0	32	0	1
● lowerBoundMinSupportTipText()		0%		n/a	1	1	1	1	1	1
● main(String[])		0%		n/a	1	1	2	2	1	1
● metricString()		0%		0%	4	4	5	5	1	1
● metricTypeTipText()		0%		n/a	1	1	1	1	1	1
● mineCARs(Instances)		0%		n/a	1	1	3	3	1	1
● minMetricTipText()		0%		n/a	1	1	1	1	1	1
● numRulesTipText()		0%		n/a	1	1	1	1	1	1
● outputItemSetsTipText()		0%		n/a	1	1	1	1	1	1
● pruneRulesForUpperBoundSupport()		0%		0%	5	5	21	21	1	1
● removeAllMissingColsTipText()		0%		n/a	1	1	1	1	1	1
● removeMissingColumns(Instances)		0%		0%	12	12	34	34	1	1
● resetOptions()		100%		n/a	0	1	0	14	0	1
● setCar(boolean)		0%		n/a	1	1	2	2	1	1
● setClassIndex(int)		0%		n/a	1	1	2	2	1	1
● setDelta(double)		0%		n/a	1	1	2	2	1	1
● setLowerBoundMinSupport(double)		0%		n/a	1	1	2	2	1	1
● setMetricType(SelectedTag)		82%		50%	5	6	2	9	0	1
● setMinMetric(double)		100%		n/a	0	1	0	2	0	1
● setNumRules(int)		0%		n/a	1	1	2	2	1	1
● setOptions(String[])		93%		86%	3	12	4	38	0	1
● setOutputItemSets(boolean)		0%		n/a	1	1	2	2	1	1
● setRemoveAllMissingCols(boolean)		100%		n/a	0	1	0	2	0	1
● setSignificanceLevel(double)		0%		n/a	1	1	2	2	1	1
● setTreatZeroAsMissing(boolean)		0%		n/a	1	1	2	2	1	1
● setUpperBoundMinSupport(double)		100%		n/a	0	1	0	2	0	1
● setVerbose(boolean)		0%		n/a	1	1	2	2	1	1
● significanceLevelTipText()		0%		n/a	1	1	1	1	1	1
● static {...}		100%		n/a	0	1	0	3	0	1
● toString()		52%		37%	25	28	73	138	0	1
● treatZeroAsMissingTipText()		0%		n/a	1	1	1	1	1	1
● upperBoundMinSupportTipText()		0%		n/a	1	1	1	1	1	1
● verboseTipText()		0%		n/a	1	1	1	1	1	1
Total	1,548 of 3,426	54%	149 of 246	39%	153	194	347	679	51	67

Annexe_Figure 5: Couvrage de l'algorithme « Apriori »