

# École Polytechnique de Montréal

Département de Génie Informatique et Génie Logiciel



## **LOG8371 INGÉNIERIE DE LA QUALITÉ EN LOGICIEL HIVER 2019**

### **Plan de tests**

Travail pratique no 1  
Groupe no 1

Soumis par  
Équipe Mike

29 février 2019

# Table des matières

<b>Introduction</b>	<b>2</b>
<b>Portée des tests</b>	<b>2</b>
<b>Environnement d'utilisation des tests</b>	<b>2</b>
Emplacement	2
Configuration matérielle	2
Parties prenantes	2
Exigences de personnel	3
Préparation et entraînement de l'équipe	3
<b>Description des cas de test</b>	<b>3</b>
Tests de la maintenabilité	3
Tests de fiabilité	5
Tests d'aptitude fonctionnelle	6
Tests de système	6
Tests de régression	6
Tests de performance	7
<b>Description implémentation des tests</b>	<b>8</b>
<b>Résultats des tests</b>	<b>11</b>
Tests de la maintenabilité	11
Pull Request	11
Complexité cyclomatique	13
Tests de fiabilité	13
Rapport d'erreur avec Rollbar	13
Tests d'aptitude fonctionnelle - Couverture de code avec tests unitaires	14
<b>Bibliographie</b>	<b>16</b>

# Introduction

Au travers de ce document nous allons présenter les tests qui seront utilisés pour valider les différents objectifs des critères de qualités qui ont été annoncé dans le plan d'assurance qualité.

## Portée des tests

Puisque le système Weka est un logiciel accessible à tous, nous nous sommes assuré de couvrir le plus d'algorithmes variés possible. Donc dans le cadre de ce travail nous avons choisi de concentrer nos efforts sur les algorithmes suivants et leurs fichiers associés:

- 1) Algorithme d'associations présent dans le fichier *Apriori.java*
- 2) Algorithme d'associations présent dans le fichier *FPGrowth.java*
- 3) Algorithme de regroupement présent dans le fichier *Cobweb.java*
- 4) Algorithme de regroupement présent dans le fichier *FarthestFirst.java*
- 5) Algorithme de classifications présent dans le fichier *NaiveBayes.java*

## Environnement d'utilisation des tests

### Emplacement

Tel que mentionné précédemment les tests se concentreront sur des algorithmes présents dans les cinq fichiers distincts. Les tests pour les algorithmes d'association utiliseront le contenu des fichiers *Apriori.java* et *FPGrowth.java*. Les algorithmes de regroupement seront testés à partir des fichiers *Cobweb.java* et *FarthestFirst.java*. Puis, en ce qui trait aux algorithmes de classification, ils seront testés à partir des informations contenu dans le fichier *NaiveBayes.java*.

### Configuration matérielle

Pour permettre à l'équipe de bien mener à terme ces tests, il est important que chaque membre possède un ordinateur fonctionnel capable de faire rouler des algorithmes d'apprentissage machine. Il est donc préférable que les ordinateurs soient munis d'un processeur i5 au minimum. Pour parvenir au système Weka et aux tests de fonctionner correctement chaque machine se doit également de posséder un environnement de test fonctionnel. Celui-ci se doit d'avoir les utilitaires Java d'installés en plus d'avoir un environnement de développement (IDE) spécialisé en Java.

### Parties prenantes

Les parties prenantes seront les mêmes que ceux mentionnés dans le plan d'assurance qualité.

## Exigences de personnel

Pour mener à terme ce plan de tests, il faut se munir d'une bonne équipe de testeur. En plus d'avoir des connaissances dans le domaine de l'intelligence artificielle, le machine learning et l'informatique en général, ces testeurs doivent également avoir une bonne vision globale du projet. En général une équipe de cinq membres permettrait de remplir les requis de ce plan de tests.

## Préparation et entraînement de l'équipe

Tel que mentionné plus haut, pour préparer les environnements de travail ainsi que l'équipe il faut s'assurer que tous ait une connaissance du langage Java. De plus, puisque le système Weka est très utilisé dans le domaine de l'intelligence artificielle il est important que les testeurs acquièrent des connaissances sur ce domaine. Ainsi donc, il faut entraîner l'équipe à procéder au traitement d'informations avec les différents algorithmes présents dans le système Weka. De cette façon, l'équipe s'entraîne à comprendre les besoins des utilisateurs et s'approprie le système le plus possible.

## Description des cas de test

Dans la présente section nous allons présenter en détails les différents tests qui permettront de tester correctement les différentes exigences du système Weka ainsi que les différents critères de qualités qui ont été définis dans le plan d'assurance qualité. Cette liste détaillée comportera des tests de différents niveaux. De cette façon, les tests unitaires, d'intégration, de système, etc. couvriront la majorité des catégories testables du système. Pour faciliter la présentation des différents tests, ils seront séparés selon leur type et seront présentés sous forme de tableaux descriptifs.

## Tests de la maintenabilité

Tel que défini dans le plan d'assurance qualité, la maintenabilité représente le degré d'efficacité et d'efficience avec lequel un produit peut être modifié par les responsables de la maintenance. Pour le bien de ce travail l'équipe de développement a choisi de se concentrer sur les sous-critères de la simplicité et de la modularité du système. Cela signifie donc que les différents tests couvrant la simplicité du système doivent vérifier que le code est facile à comprendre et à lire. En ce qui trait à la modularité, les différents tests mis en place doivent vérifier que le logiciel possède un niveau de couplage assez bas entre les différentes classes ou modules. La majorité des tests qui permettent de couvrir ces deux sous-catégories sont beaucoup plus subjectifs puisqu'ils se rapportent plus à l'idée de mettre en place de bonnes normes et pratiques de codage au sein de l'équipe de développement. Les tests présentés ci-dessous permettent donc de vérifier que ces critères sont respectés et par le fait même permettent à l'équipe d'instaurer différentes règles de codage.

<b>Objectif de test:</b>	Maintenir une simplicité de code
<b>Technique:</b>	Instauration de revue de code régulières entre les différents membres de l'équipe de développement. Ces techniques peuvent avoir lieu à chaque demande de mise en commun de code (" <i>pull request</i> ").
<b>Critère de complétion:</b>	Le code doit être facilement lisible et compréhensible et ne doit pas comporter de fautes graves.
<b>Considérations spéciales:</b>	L'instauration de revue de code plus important lors des différents jalons importants du projet serait également un bon ajout.

<b>Objectif de test:</b>	Réduire la complexité cyclomatique du code
<b>Technique:</b>	Afin de réduire la complexité cyclomatique du système le plus bas possible, l'équipe de développement doit utiliser un outil évaluateur comme JaCoCo ou SonarQube.
<b>Critère de complétion:</b>	La complexité cyclomatique du code doit être maintenue en bas de 25.
<b>Considérations spéciales:</b>	Aucune.

<b>Objectif de test:</b>	Avoir un code de basse modularité en assurant une dépendance minimale entre paquetage du système.
<b>Technique:</b>	Calculer le niveau de modularité du système en faisant un diagramme de paquetage avant chaque fin de jalon. Avec ce diagramme, calculer l'indice de modularité du système.
<b>Critère de complétion:</b>	Suite à son calcul l'indice de modularité (présenté dans le plan d'assurance qualité) ne doit pas dépasser 0.4.
<b>Considérations spéciales:</b>	Aucune

## Tests de fiabilité

Dans le cadre du système Weka, la fiabilité du système sera testé selon les deux sous-critères qui ont été définis dans le plan d'assurance qualité soit: la récupérabilité ainsi que la tolérance aux fautes et aux échecs. Les tests suivants se concentreront donc à vérifier que le système peut facilement être rétabli après une panne et que le nombres d'échec reportés est le plus bas possible.

<b>Objectif de test:</b>	Le but de ce tests est de s'assurer que le logiciel est capable de récupérer suite à une défaillance sans compromettre l'intégrité de ses donnés. (Test de robustesse)
<b>Technique:</b>	Forcer le système a traité une grande quantité de donnés afin de causer un échec ou une panne.
<b>Critère de complétion:</b>	Le temps de récupération du système ne doit pas dépasser 60 secondes.
<b>Considérations spéciales:</b>	Ce test doit être également être fait avant chaque lancement d'une nouvelle version du logiciel.

<b>Objectif de test:</b>	Vérifier que le temps moyens entre les différents échecs reportés par les utilisateurs est le plus court possible.
<b>Technique:</b>	À chaque erreur ou panne reporté par un utilisateur un rapport est généré dans un système de l'entreprise. À partir de ses rapports différentes métriques sont générées ce qui permet à l'équipe de développement de savoir quelles composantes sont problématiques et doivent être priorisées. L'outil utilisé pour générer les rapports est Rollbar.
<b>Critère de complétion:</b>	Le temps moyens entre les différents échecs reportés par les utilisateurs doit être supérieur à 5 jours dans une période de 1 mois.
<b>Considérations spéciales:</b>	Les nouvelles versions du logiciel ne doivent pas causer des anciens problèmes de surgir. Tous les tests unitaires et d'intégration qui passaient dans une version précédente doivent passer dans les nouvelles versions.

## Tests d'aptitude fonctionnelle

Les différents tests d'aptitude fonctionnelle sont utiles afin de tester les différentes fonctionnalités du système Weka. Ces tests s'assureront donc que le système est capable d'exécuter des tâches selon les normes spécifiées. Comme le mentionne le document de plan d'assurance qualité l'équipe a choisi de tester les deux sous-critères suivants: la complétude fonctionnelle et la précision fonctionnelle. Puisque les différents algorithmes du système Weka sont subdivisés sous différentes catégories il est important que les algorithmes associés à une même catégorie répondent aux mêmes besoins. Par exemple, un algorithme faisant partie de la catégorie d'algorithmes d'association doit pouvoir être capable de générer les règles indépendamment de la méthode.

Pour tester la complétude fonctionnelle nous avons choisi de faire usage des différents tests unitaires déjà disponibles avec le système Weka. Puisque la liste est très très longue et qu'elle est disponible avec la documentation du système nous avons jugé qu'il n'était pas nécessaire de la reproduire ici. Toutefois, puisqu'il est demandé d'ajouter un nouvel algorithme au système et que nous avons choisi d'ajouter un algorithme d'association les différents tests unitaires qui sont associés à cette catégorie seront énoncés dans la prochaine section nommée *"Implémentations des tests - Tests d'aptitude fonctionnelle"*.

Pour tester la seconde sous-catégorie *"Précision fonctionnelle"* nous devons vérifier que les différents tests unitaires prédéfinis par Weka, mentionnés ci-dessus, obtiennent un taux de succès d'environ 90%. Tel que mentionné dans le plan d'assurance qualité, nous savons qu'il est peut-être probable d'atteindre un taux de succès de 100% puisque Weka est un système possédant des algorithmes d'apprentissage machine et qu'il travaille constamment avec d'énormes données. C'est pourquoi nous énonçons que si le taux de réussite des tests unitaires et d'intégration est plus que 90% nous avons atteint l'objectif.

## Tests de système

Puisque Weka définit également des tests systèmes en plus des différents tests unitaires pour les différentes catégories d'algorithmes nous ne pensons pas ajouter d'autres cas de tests supplémentaires. Pour la même raison qu'énoncé à la section précédente nous n'allons pas tous les énumérer ici. Toutefois, en ce qui trait à leur critère de complétion nous allons maintenir le taux de réussite de 90%. De plus, pour que les tests soient concluants, il faut également que chaque test passe et qu'il n'y ait aucune erreur.

## Tests de régression

Puisque nous souhaitons que le système passe constamment toutes les suites de tests qui ont été définies dans les sections précédentes, le premier groupe de tests que les tests de régression devront couvrir sont ceux-ci. Afin d'avoir un succès aucun des anciens tests ne doivent échouer.

Ces tests seront donc lancés à chaque fois que le système est modifié et avant un déploiement ou un livrable.

## Tests de performance

Les tests de performance sont très utiles pour déterminer les limites d'un système. En quelques grandes lignes ils permettent de déterminer les limites de temps, de ressources et de capacité. Dans le cadre du système Weka, la performance du système sera testée et vérifiée par les trois sous-critères qui ont été définis dans le plan d'assurance qualité soit : le comportement du système par rapport au temps, l'utilisation des ressources et la capacité du système. Les tests suivants se concentreront donc à vérifier que le système ne consomme pas trop de ressource et qu'il est capable de fonctionner de manière raisonnable avec un temps de réponse restreint.

<b>Objectif de test:</b>	Vérifier que le système prend moins de 3 minutes pour traiter un fichier d'au moins 55 Mo.
<b>Technique:</b>	Nous allons lancer différentes tâches avec des fichiers de taille variable et calculer leur temps de traitement. Il ne doit pas excéder 3 minutes si la taille est inférieure à 55 Mo.
<b>Critère de complétion:</b>	Le délais de traitement respectant la contrainte de taille doit être moins de 3 minutes.
<b>Considérations spéciales:</b>	Aucune

<b>Objectif de test:</b>	Vérifier que le système doit utiliser au maximum 1 Go de mémoire vive pour un fichier d'environ 55 Mo.
<b>Technique:</b>	Nous allons lancer différentes tâches avec des fichiers de taille variable autour de 50 Mo et calculer le taux d'utilisation de la mémoire vive du système. Il ne doit pas excéder 1Go si la taille du fichier est inférieure à 55 Mo.
<b>Critère de complétion:</b>	Le taux d'utilisation de ressource doit être en deçà de 1Go.
<b>Considérations spéciales:</b>	Il est toutefois attendu que le système utilise la majorité de ses capacités pour effectuer ses tâches.



<b>Objectif de test:</b>	Vérifier que le système n'est pas à sa capacité limite lorsque l'on traite des fichiers ayant une grosseur de 55 Mo.
<b>Technique:</b>	Nous allons lancer différentes tâches avec des fichiers de taille variable et calculer leur pourcentage d'utilisation du CPU. Il ne doit pas excéder 100%. De plus, il doit se retrouver autour d'une valeur approximative de 60% lorsque le système traite des fichiers de 55 Mo.
<b>Critère de complétion:</b>	Le pourcentage d'utilisation du CPU ne doit absolument pas dépasser 100% et doit se situer autour de 60% d'utilisation.
<b>Considérations spéciales:</b>	Ce test permet de déterminer les limites maximales du système. Donc tout dépendant du système qui roule Weka ces limites peuvent varier.

## Description implémentation des tests

Dans cette section, nous décrivons les tests de code présent dans Weka ainsi que ceux que nous avons rajouté. Nous n'avons pas eu à implémenter d'autres tests pour les autres critères étant donné que les autres tests ne font qu'utiliser des outils déjà existants. Cependant, les résultats de tous les types de tests sont présents dans la section suivante.

### Tests d'aptitude fonctionnelle

Les tests décrits dans la section suivante sont utilisés pour tester tous les algorithmes de type Associations du projet Weka. Comme nous avons ajouté un algorithme de ce type, notre nouvel algorithme sera sujet aux tests communs à cette catégorie. De plus, il a fallu ajouter des tests spécifiques à notre algorithme pour s'assurer que ses méthodes aient le comportement désiré. Comme notre algorithme ne comporte qu'une seule méthode, 'getGlobalInfoHelloWorld()', nous n'avons eu qu'à ajouter un seul test supplémentaire (il a fallu ajouter l'implémentation de ce test à tous les algorithmes qui ont hérité de la classe AbstractAssociation)

Méthode	Retour	Description
canPredict(int type)	boolean	checks whether at least one attribute type can be handled with the given class type

getClassTypeString(int type)	String	returns a string for the class type
testAttributes()	void	tests whether the Associator can handle different types of attributes
testSerialVersionUID()	void	tests whether the scheme declares a serialVersionUID
testInstanceWeights()	void	tests whether the Associator handles instance weights correctly
testNClasses()	void	tests whether Associator handles N classes
checkClassAsNthAttributes()	void	checks whether the Associator can handle the class attribute given
testZeroTraining()	void	tests whether the Associator can handle zero training instances
testMissingPredictors()	void	checks whether the Associator can handle the given percentage of missing predictors
testMissingClass()	void	tests whether the Associator can handle missing class values
testBuildInitialization()	void	tests whether the Associator correctly initializes in the buildAssociator method
testDatasetIntegrity()	void	tests whether the Associator

		alters the training set during training
useAssociator(Instances data)	String	Builds a model using the current Associator using the given data and returns the produced output
testRegression()	void	Runs a regression test
testGlobal()	void	tests for a globalInfo method
testGlobalInfoHelloWorld()	void	tests for a globalInfoHelloWorld method

**Tableau 1** : Suite de tests du groupe d'algorithme Associations

# Résultats des tests

## Tests de la maintenabilité

### *Pull Request*

Voici à quoi ressemble un pull request sur Github.

## Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also [compare across forks](#).


base repository: ecgrimard/weka-3.8 ▾

base: master ▾

head repository: ChloeConstantineau/weka-3.8 ▾

compare: master ▾

✓ **Able to merge.** These branches can be automatically merged.



Adding travis ci integration

Write

Preview

AA

B

i

“

<>

↻

: :

≡

≡

≡

@

🔖

↶

Read updated [read me](#) for more info

Attach files by dragging & dropping, selecting them, or pasting from the clipboard.

☒ Allow edits from maintainers. [Learn more](#)


Create Pull Request

▾

Reviewers

No reviews

Assignees

 ecgrimard

Labels

None yet

Projects

None yet

Milestone

No milestone

**Figure 1:** Interface de création d'une “pull request”

## Adding travis ci integration #1

[Edit](#)[Open](#)ChloeConstantineau... wants to merge 25 commits into `ecgrimard:master` from `ChloeConstantineau:master`

Conversation 0

Commits 25

Checks 0

Files changed 11

+2,140 -23



ChloeConstantineau... commented just now

Collaborator

+ 👤 ...

Read updated read me for more info

ChloeConstantineau added some commits 27 days ago

- Create test.travis.yml
- Update test.travis.yml
- Update test.travis.yml
- Update and rename test.travis.yml to .travis.yml
- Update .travis.yml
- Update .travis.yml
- Update .travis.yml
- Update .travis.yml

Verified	746cc24
Verified	5855b17
Verified	a716439
Verified	665bb6d
Verified	85bdbea
Verified	efdc62b
Verified	c449d18
Verified	f82c1a5

Reviewers

No reviews

Assignees

ecgrimard

Labels

None yet

Projects

None yet

Milestone

No milestone

Figure 2: Interface visuelle de la demande de mise en commun

Puisque le contenu de la branche ne comporte aucuns conflits ou problème, nous pouvons procéder.

ChloeConstantineau assigned ecgrimard just now

Add more commits by pushing to the `master` branch on `ChloeConstantineau/weka-3.8`.

**This branch has no conflicts with the base branch**  
Merging can be performed automatically.

[Merge pull request](#) You can also [open this in GitHub Desktop](#) or view [command line instructions](#).

Figure 3: Représentation visuelle permettant de procéder à la mise en commun

### Complexité cyclomatique

Voici une représentation de la complexité cyclomatique d'une fonction de l'algorithme Apriori

Code analyzed successfully.


File Type **.java** Token Count **1196** NLOC **129**

Function Name	NLOC	Complexity	Token #	Parameter #
buildAssociations	129	31	1194	







**Figure 4:** Représentation de l'analyse cyclomatique de la fonction Apriori

### Tests de fiabilité

Rapport d'erreur avec Rollbar

FRAMEWORK	ENVIRONMENT	OWNER	STATUS
 Java	Show All	Show All	Active

Last ↓	Item
59 secs	 #170672 HandlerCommon - QUEUE_PROCESSING_RESULT: id=265657, env=p5.zrh, s...
1 min	 Comp ORIGIN_IS_NOT_SET: Origin is not set oxm2RequestParams(vertical=Flight, oac...
2 mins	 IMAGE_RESIZE_PARAMETER_VALIDATION_FAILED: reason=UtilsMessage.INVALID_IMA...
3 mins	 Search - UNABLE_ATTACH_GROUP_EMPTY: Failed to attach ID. hid=3ffad920, ctid=1...
3 mins	 AccessDeniedException - HANDLE_UNEXPECTED: com.demoxfactor.framework.Acce...
6 mins	 #428081 ZoneRulesException - HANDLE_UNEXPECTED: java.time.zone.ZoneRulesExc...

**Figure 5:** Capture d'écran présentant l'interface du rapport d'erreur produit par Rollbar

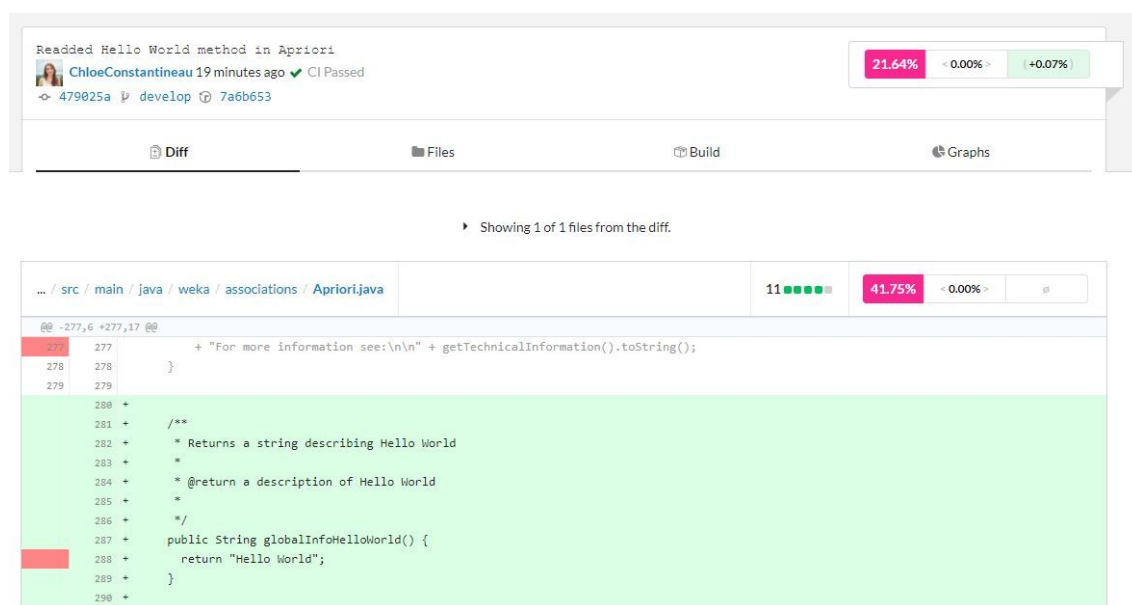
## Tests d'aptitude fonctionnelle - Couverture de code avec tests unitaires

Les résultats de tous les tests unitaires sont présentés dans le fichier *RésultatsTests.txt*. Pour en faire un résumé, un total de 4094 tests ont été effectués avec aucune erreur, aucun échec et aucun test reporté. Il leur a fallu une minute et 43 secondes pour compléter avec succès. Chaque suite de tests a été exécutée. Nous pouvons remarquer que le taux de couverture de notre code n'est pas à 100%. Il aurait fallu écrire une quantité substantielle de code pour y arriver à temps pour la remise. Cependant, l'équipe devra dédier un jalon du projet pour écrire des tests et augmenter la couverture de code à un plus haut niveau.

Comme mentionné dans le plan d'intégration continue, nous avons ajouté l'outil CodeCov pour obtenir une couverture de code effectuée grâce aux tests. Pour avoir une vue d'ensemble de la couverture de code au moment du build mentionné précédemment, il est possible de visiter le site web, généré lors de l'exécution des tests, suivant :

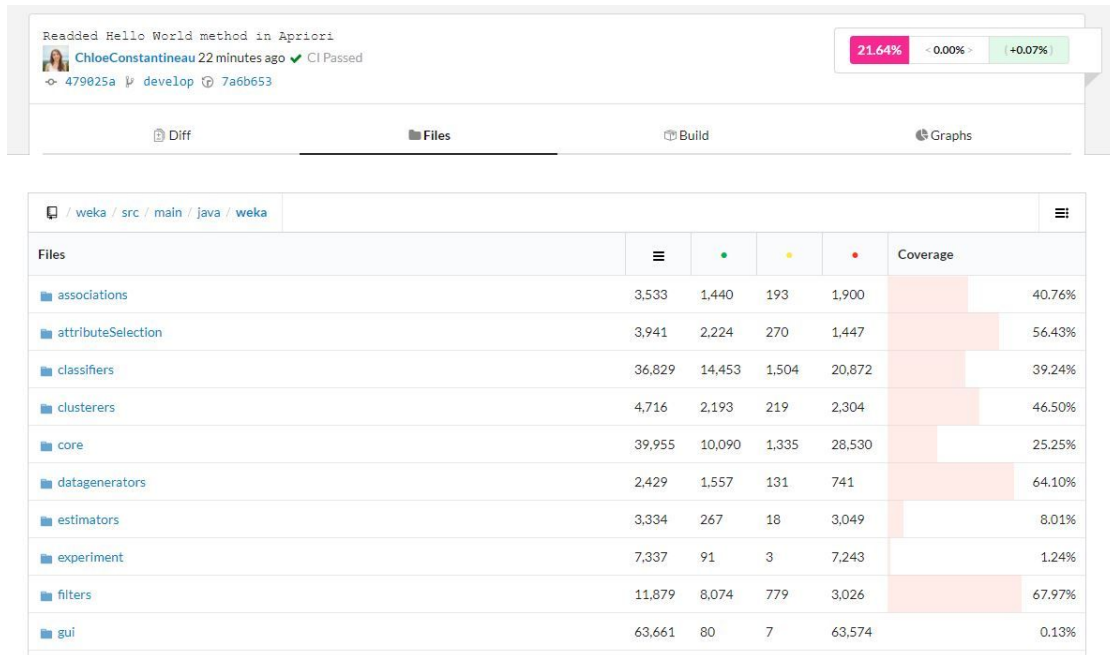
<https://codecov.io/gh/ChloeConstantineau/weka-3.8/tree/develop/weka/src/main/java/weka>

Voici un aperçu de ce qu'on y retrouve :



**Figure 6 :** Méthode unique de notre algorithme ajouté

Cette capture d'écran indique en vert la méthode ajoutée dans le code. La couleur vert indique que cette méthode est couverte par un test et que ces lignes de code sont bien traversées. On peut y voir aussi en haut à gauche la couverture de code du projet qui est à 21,34% et qui a augmenté de 0,07% avec le commit en question. On remarque que le fichier en tant que tel a pour sa part une couverture de code de 41,75%.



**Figure 7 :** Couverture de code de toutes les catégories d'algorithme



# Bibliographie

[1] Bouchard Valérie, Constantineau Chloé, Grimard Éric, Plante Roxanne. ***Plan d'assurance qualité***. LOG8371. 2019