

École Polytechnique de Montréal
Département de Génie informatique et Génie logiciel

LOG8371
Ingénierie de la qualité en logiciel

Travail Pratique 1
Plan et test

Soumis à: monsieur mystère

Soumis par
Antoine Boisier-Michaud (1792089)
Rémi Pelletier (1788041)
Simon Lacasse (1789986)
Mikaël LeBlanc (1796758)
Michaël Sghaïer (1725355)

21 février 2019

Table des Matières

Table des Matières	2
Question 1	3
Introduction	3
Exigences fonctionnelles	3
Critères de qualité	4
Fonctionnalité	4
Complétude	4
Fiabilité	5
Maintenabilité	6
Plan de vérification	7
Validation des objectifs	7
Stratégie d'intégration continue	7
Stratégies de revue et d'inspection	7
Question 2	8
Plan de validation des objectifs	8
Fonctionnalité	8
Complétude	8
Exactitude	8
Convenance	9
Fiabilité	10
Exactitude	10
Maintenabilité	10
Question 3	11
Artefacts (intrants)	11
Artéfacts (extrants) et preuve d'exécution	11
Question 4	13
Question 5	14
Sources	15

Question 1

Introduction

Weka est une collection d'algorithmes de forage et d'interprétation de données massives. Cette librairie contient des outils de préparation, de classification, de régression, d'agrégation, de forage de règles d'associations et de visualisation de données. Les applications de l'apprentissage machine sont nombreuses, allant du domaine de la santé, aux marchés boursiers, en passant par le secteur manufacturier et l'industrie du divertissement. En règle générale toutes les compagnies voulant extraire de l'information d'ensembles massifs de données sont des clients potentiels de Weka. Les qualités attendues sont la Fonctionnalité, la Fiabilité et la Maintenabilité.

Exigences fonctionnelles

La version de la librairie Weka que nous allons explorer dans le cadre de ce travail devra être capable d'exécuter cinq algorithmes, soit deux classificateurs, deux algorithmes d'agrégation, et un algorithme d'association. L'algorithme d'association sera FPGrowth, qui est un algorithme permettant de trouver les *patterns* les plus fréquents dans un ensemble de données. Comme algorithme de classification, nous incluons les algorithmes NaiveBayes et RandomForest. NaiveBayes est un algorithme de classification probabiliste simple basé sur le théorème de Bayes. RandomForest est une méthode de classification et de régression se basant sur les arbres de décision. Deux algorithmes d'agrégation seront aussi inclus dans notre version réduite de la librairie Weka. Ces deux derniers seront K-Means et FarthestFirst. L'algorithme K-Means permet de partitionner un ensemble de données dont le nombre de catégories est connu d'avance en initialisant aléatoirement un nombre K de centroïdes qui sont itérativement perfectionnés. La méthode d'agrégation Farthest First utilise une traversée de K points d'un ensemble de données dans un ordre maximisant une mesure de distance afin d'optimiser la répartition des agrégats.

Critères de qualité

Les critères de qualité seront décrits dans cette section. La fonctionnalité, la fiabilité et la maintenabilité seront abordés, ainsi que leurs sous-critères. Pour l'évaluation des métriques de nos critères de la qualité, nous utiliserons Code Climate. Ce logiciel évalue la qualité du code et fournit certaines métriques, en plus de donner une note de A à F sur les critères. Nos objectifs seront parfois basées directement sur la métrique, parfois basées sur la note donnée.

Fonctionnalité

À tous les stades, il faut s'assurer que le logiciel est complètement fonctionnel dans tous ses états possibles et dans toutes ses zones.

Complétude

La complétude consiste en le caractère complet des fonctionnalités. Ainsi, les fonctionnalités désirées comportent toutes leurs parties nécessaires et ne manquent rien pour leur bon fonctionnement.

Métriques :

- Pour tester la complétude des fonctionnalités, nous utiliserons les tests fournis. Nous désirons qu'au moins un test unitaire relié à la complétude par algorithme soit réussi.

Exactitude

L'exactitude consiste en le caractère exact des résultats donnés par la fonctionnalité. La fonctionnalité doit retourner le résultat attendu et correct pour être considéré comme exact. Dans notre cas, la nature des algorithmes étudiés préviennent la mesure d'un tel critère. En effet, plusieurs de ces algorithmes ne sont pas complètement déterministes, ce qui nous empêche d'avoir une réponse exacte à tous les coups. Pour ceux qui le sont, ce critère sera aussi évalué par les tests évaluant la cohérence.

Convenance

La convenance représente la conformité à un certain standard ou à un objectif. Une attente est définie envers la fonctionnalité et elle doit être respectée pour être marquée comme convenable.

Métriques :

- Pour la convenance des fonctionnalités, nous utiliserons les tests déjà fournis.

Fiabilité

La fiabilité indique la capacité du logiciel à ne pas entrer dans un état d'erreur lors de son exécution et de fonctionner tel que désiré.

Cohérence

La cohérence représente le fait que les fonctionnalités retournent toujours le même résultat lorsque les mêmes intrants leurs sont fournis.

Métriques :

- Pour la cohérence, nous utiliserons les tests déjà fournis. Nous désirons qu'au moins un test unitaire relié à la cohérence par algorithme soit réussi.

Précision

La précision consiste à donner le bon résultat tout en étant dépourvu d'erreurs. Dans notre cas, la nature des algorithmes étudiés préviennent la mesure d'un tel critère. En effet, plusieurs de ces algorithmes ne sont pas complètement déterministes, ce qui nous empêche d'avoir une réponse exacte à tous les coups. Pour ceux qui le sont, ce critère sera aussi évalué par les tests évaluant la cohérence.

Tolérance aux erreurs

La capacité du logiciel à tolérer les erreurs, c'est-à-dire à bien réagir aux erreurs survenues pour tout de même donner la bonne réponse ou, à tout le moins, réagir gracieusement.

Métriques :

- Nous utiliserons des tests unitaires avec des intrants erronés pour tester la tolérance aux erreurs.

Maintenabilité

La maintenabilité indique la facilité avec laquelle un système peut évoluer pour proposer de nouvelles fonctionnalités ou résoudre des bogues. Ce critère peut être caractérisé par les sous-critères suivants :

Simplicité

La simplicité est la qualité d'être simple et non complexe. Rapporté à un système, elle indique que la solution mise en œuvre pour résoudre un problème s'exprime de façon claire et simple, compréhensible en une rapide étude.

Métriques :

- Nous utiliserons, comme métrique de la simplicité, la complexité cyclomatique. Nous désirons que les méthodes aient une complexité cyclomatique maximale de n^3

Concision

La concision est une propriété qui se rapporte à la taille de la solution mise en œuvre. Une solution est dite concise quand celle-ci est de taille minimale, relativement au contexte du problème, et qu'elle se débarrasse de tout élément superflu.

Métriques :

- Les lignes de codes seront notre critère pour évaluer la concision. Nous désirons que les méthodes ne dépassent pas 100 lignes de code et que les classes ne dépassent pas 1000 lignes de code

Description réflexive

La description réflexive est la propriété d'une solution à se passer d'explications. La solution se suffit à elle-même et sa simple lecture permet de comprendre sa logique intrinsèque.

Métriques :

- Les mauvaises odeurs seront une métrique d'évaluation de la description réflexive. Nous désirons avoir la note B dans la catégorie des mauvaises odeurs.

Modularité

La modularité est la division d'un logiciel plus complexe en plusieurs modules plus petits qui sont plus faciles à maintenir de façon indépendante.

Métriques :

- Pour l'évaluation de la modularité, nous évaluerons

Plan de vérification

Validation des objectifs

Pour déterminer si les objectifs ont été atteints, il sera possible de vérifier des indicateurs sur les tableaux de bord de Travis Ci et Code Climate seront consultés. Des alertes seront également lancées par courriel lorsque les objectifs ne sont pas atteints.

Stratégie d'intégration continue

Pour l'intégration continue, nous avons décidé d'utiliser Travis Ci et Code climate pour notre intégration continue. Lorsque nous poussons une nouvelle version sur notre répertoire GitHub, Travis Ci et Code Climate clonent le dépôt automatiquement et effectuent certaines analyses.

Travis Ci compile le code et roule ensuite les tests. Lorsque le build et les tests sont complétés, Travis Ci envoie une notification par courriel à l'auteur de la nouvelle version pour l'informer du statut du *build*. De cette manière, les développeurs savent si le *build* est brisé ou non.

Code Climate, quant à lui, analyse la structure du code, détecte les mauvaises odeurs et produit un rapport sous forme de tableau de bord. Le rapport indique le nombre de mauvaises odeurs détectées, où elles se trouvent et le temps requis pour corriger toute la dette technique accumulée dans le code.

Stratégies de revue et d'inspection

Chaque nouvelle version devra être revue et approuvée par un minimum de 3 personnes. Pour ce faire, les développeurs devront *forker* le dépôt et travailler sur leur propre *fork*. Lorsque la nouvelle version est prête, une demande de tirage devra être effectuée depuis GitHub. Il est impossible de fusionner la *fork* avec la branche principale (*master*) sans avoir obtenu trois

approbations. Si la nouvelle version n'est pas approuvée, le responsable doit appliquer les suggestions proposées par les réviseurs et attendre que ceux-ci fasse une seconde révision. On répète ce processus jusqu'à ce que la version soit approuvée.

Question 2

Plan de test et suites de tests

Étant donné la taille très petite de notre version de la librairie weka et en considérant le fait que nous créons essentiellement une version réduite d'une librairie existante, nous avons décidé d'utiliser la stratégie de tests "Big Bang" lors des étapes de validation précédant une release. De façon similaire, la petite taille de notre jeu de tests unitaires fait en sorte qu'une seule suite contenant tous les tests unitaires sera nécessaire. Les tests unitaires inclus à cette suite sont tous détaillés dans la section suivante, qui explique aussi comment ces tests valideront les objectifs spécifiés à la question 1.

Plan de validation des objectifs

Fonctionnalité

Complétude

Une version du test suivant est appliqué à toutes les classe d'algorithme incluses dans notre librairie. Il est situé dans les classes `AbstractAssociatorTest`, `AbstractClassifierTest` et `AbstractClustererTest`.

```
/**
 * tests whether the classifier can handle different types of attributes
 */
public void testAttributes() {
    // nominal
    checkAttributes(true, false, false, false, false, true);
    // numeric
    checkAttributes(false, true, false, false, false, true);
    // string
    checkAttributes(false, false, true, false, false, true);
}
```



```

// date
checkAttributes(false, false, false, true, false, true);
// relational
if (!m_multiInstanceHandler) {
    checkAttributes(false, false, false, false, true, true);
}
}

```

Exactitude

Comme mentionné plus haut, il ne nous est pas possible d'évaluer ce critère pour tous les algorithmes. Pour ceux pour lesquels il est possible de le faire (FPGrowth et NaiveBayes), le test de Cohérence nous sera suffisant afin de les valider aussi

Convenance

Une version du test suivant est appliqué à toutes les classe d'algorithme incluses dans notre librairie. Il est situé dans les classes AbstractAssociatorTest, AbstractClassifierTest et AbstractClustererTest. Ce test évalue si les différentes classes sont toujours fonctionnelle avec différents pourcentages de leur prédicteurs manquants.

```

/**
 * tests whether the classifier can handle missing predictors (20% and 100%)
 */
public void testMissingPredictors() {
    int i;

    for (i = FIRST_CLASSTYPE; i <= LAST_CLASSTYPE; i++) {
        // does the classifier support this type of class at all?
        if (!canPredict(i)) {
            continue;
        }

        // 20% missing
        checkMissingPredictors(i, 20, true);

        // 100% missing
        if (m_handleMissingPredictors[i]) {
            checkMissingPredictors(i, 100, true);
        }
    }
}

```

Fiabilité

Cohérence

Une version du test suivant est appliqué à toutes les classe d'algorithme incluses dans notre librairie. Il est situé dans les classes `AbstractAssociatorTest`, `AbstractClassifierTest` et `AbstractClustererTest`. Dans tout le cas, il vérifie si une valeur nouvellement calculé correspond à une valeur calculé à une itération antérieur.

```
/**
 * Runs a regression test -- this checks that the output of the tested object
 * matches that in a reference version. When this test is run without any
 * pre-existing reference output, the reference version is created.
 */
public void testRegression() throws Exception {
```

Précision

Comme mentionné plus haut, il ne nous est pas possible d'évaluer ce critère pour tous les algorithmes. Pour ceux pour lesquels il est possible de le faire (FPGrowth et NaiveBayes), le test de Cohérence sera suffisant afin de les valider aussi

Tolérance aux erreurs

Une version du test suivant est appliqué à toutes les classe d'algorithme incluses dans notre librairie. Il est situé dans les classes `AbstractAssociatorTest`, `AbstractClassifierTest` et `AbstractClustererTest`. Ce test évalue si les différentes classes sont toujours fonctionnelle avec différents pourcentages de leur classes manquantes.

```
/**
 * tests whether the classifier can handle missing class values(20% and 100%)
 */
public void testMissingClasses() {
```

Maintenabilité

Les critères de maintenabilité, par leur caractère plus abstrait et à cause du fait qu'ils décrivent le code lui-même plutôt que ses fonctions, ne sont pas testables à l'aide de tests unitaires comme les autres critères le sont. Cependant, en utilisant des outils de diagnostic du code comme ptidej, il est possible de tirer des métriques du codes, et en spécifiant des échelles pour ces métriques, on peut évaluer les critères de maintenabilité. les métriques ainsi que leurs barèmes ont été spécifiés dans la section critères de qualités, et leur évaluation se fera à l'aide de ptidej

Tests de régression

La librairie de base Weka possède déjà des tests de régression pour chacun des types d'algorithmes contenus dans notre version abrégée. En les incluant à notre suite de tests qui est exécutée avant chaque compilation, nous nous assurons que les modifications que nous avons apportés durant cette phase du développement n'ont pas eu d'impact sur le résultat obtenu après l'exécution des algorithmes. Lorsque les tests de régression sont exécutés pour la première fois, la réponse est gardée en mémoire à des fins de comparaisons pour les exécutions subséquentes. Si le test est exécuté à nouveau, une nouvelle réponse est produite et les résultats sont comparés, et si les résultats sont différents, le test échouera alors, indiquant une erreur de régression. Les tests de régression sont situés dans les classes AbstractAssociatorTest, AbstractClassifierTest et AbstractClustererTest et ont la signature suivante:

```
public void testRegression() throws Exception
```

Question 3

Artefacts (intrants)

Les seuls artefacts dont Travis CI sont le code et le fichier de configuration `travis.yml`. Code Climate a seulement besoin du code. Notre fichier de configuration Travis CI peut être vu ci-contre.

```
1 language: java
2
3 before_script:
4   - cd weka
5
6 script:
7   - mvn compile
8   - mvn test
```

Artéfacts (extrants) et preuve d'exécution



L'exécution de Travis CI sur notre fork du repo de Weka donne les résultats présentés dans les captures d'écran à la page suivante. Dans l'ordre, de haut en bas, se trouvent les rapports produits par nos outils d'intégration continue.

Tout d'abord, on peut voir le tableau de bord de Travis Ci. Ce dernier affiche certaines informations relatives au dernier *build* ayant été exécuté. On peut voir qu'il a été mené à terme avec succès puisqu'il indique *passed*. On voit également le temps d'exécution, le commit à partir duquel le dépôt a été cloné, l'auteur du commit et la date d'exécution.


La seconde image présente une partie du *build log* produit par Travis Ci. Ce dernier montre les détails relatifs au *build* et aux tests. Si un ou plusieurs tests avaient échoués, c'est dans ce fichier que nous retrouverions cette information (cela est montré dans la vidéo). Dans le cas présent, on constate que 4070 tests ont été roulés, zéro ont échoué et zéro ont été ignorés. Le fichier indique que le *build* a été un succès.

La troisième capture d'écran montre le tableau de bord de Code Climate. À gauche, on peut voir le nombre de fichiers analysés. À droite, on peut voir que la note de D a été attribuée à la qualité du code. Cette note est basée sur le nombre de mauvaises odeurs qui ont été trouvées.


Ce nombre est montré au bas de la capture d'écran: 11 538. On trouve aussi le nombre d'instances de duplication de code, soit 8347. L'outil estime que la correction de toutes ces imperfections prendrait 16 années de travail pour un(e) développeur(se).


 Aboisier / weka-3.8  build passing


Current Branches Build History Pull Requests


More options 


✓ master Fixed test → #11 passed

Commit 7a3522f 

Compare 20852f4...7a3522f 


Branch master 

 Antoine Boisier-Michaud

 </> Java



```
1453 Results :
1454
1455 Tests run: 4070, Failures: 0, Errors: 0, Skipped: 0
1456
1457 [INFO] -----
1458 [INFO] BUILD SUCCESS
1459 [INFO] -----
1460 [INFO] Total time: 01:10 min
1461 [INFO] Finished at: 2019-02-12T21:37:02Z
1462 [INFO] Final Memory: 29M/326M
1463 [INFO] -----
1464 The command "mvn test" exited with 0.
1465
1466
1467
1468 Done. Your build exited with 0.
```

Top ▲

Aboisier/weka-3.8 

★ Starred

Overview Progress Issues Code Trends Repo Settings

Last  master build 4 hrs ago  Refresh

Breakdown


4,362 FILES



TEST COVERAGE

Codebase summary

MAINTAINABILITY

 16 yrs

TEST COVERAGE



Repository stats

CODE SMELLS

11,538

DUPLICATION

8,347

OTHER ISSUES

0

Le fonctionnement de ces outils est présenté plus en détail dans la vidéo.

Question 4

Comme nouvel algorithme, nous avons choisi l'algorithme d'agrégation Expectation maximization, il s'agit d'un algorithme que nous n'avions pas choisi précédemment (EM.java). Pour garantir la qualité de ce nouvel algorithme ainsi que du système entier, nous allons implémenter une suite de tests visant l'algorithme choisi. Heureusement, une série de tests pour les algorithmes d'agrégation existe déjà dans la classe `AbstractClustererTest`. Ainsi, afin d'assurer la qualité de l'algorithme Expectation maximization, il ne suffira que de créer une classe de test héritant d'`AbstractClustererTest` et d'implémenter les méthodes `getClusterer()` et `suite()` de cette nouvelle classe. L'ajout de ce nouvel algorithme ne requerra pas de mettre à jour notre plan de qualité puisqu'aucun nouveau test explicite ne devra être ajouté. De plus, notre système d'intégration continue est flexible quant l'ajout de tests et exécute ces derniers à chaque fois que du nouveau code est sur le point d'être intégré dans la branche de production.

Question 5

La vidéo suivante présente le fonctionnement de notre infrastructure CI.

Accéder à la vidéo: https://www.youtube.com/watch?v=1gNz9saA8_M

Sources

<https://pdfs.semanticscholar.org/c2c8/b9c6e7a6e4116641f2d0cd4a6691b104b38b.pdf>