HAI916I –Intelligence Artificielle et Génie Logiciel

Master 2 Informatique
TP 3 : code2vec

Ibrahim, BERKANE

ibrahim-abouyatime.berkane@etu.umontpellier.fr

Mahi, BEGOUG

mahi.begoug@etu.umontpellier.fr

1 Récupération des context-paths et leurs poids

Nous avons utilisé des expressions régulières pour récupérer les context-paths avec leurs poids.

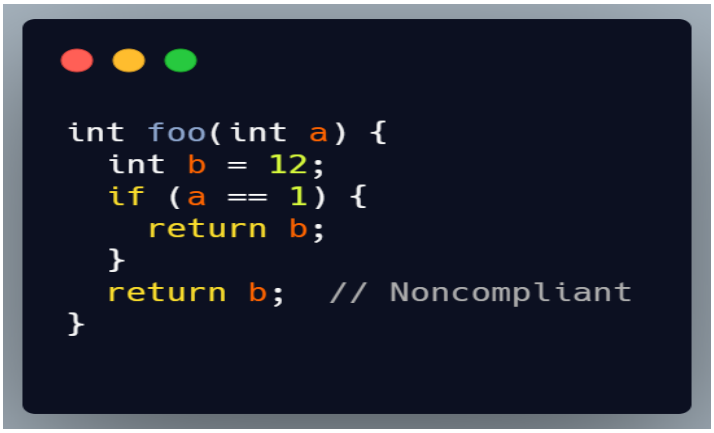
- Première étape : récupération des poids seule en utilisant cette expression régulière « [0-9]*\.[0-9]+ »
- Deuxième étape : récupération des context-path en utilisant cette expression régulière « context:.* »

2 Paths AST pour détecter les bad-smells

D'abord, nous avons étudié trois cas possibles de code semelles. Ensuite, nous avons exécuté le modèle de *code2vec* sur ces cas pour interpréter l'attention retournée. Dans ce qui suit, nous montrons les cas avec leurs attentions.

2.1 Cas 1 :

Ce cas représente lorsqu'une méthode est conçue pour renvoyer une valeur invariante, elle peut être de mauvaise conception, mais cela ne devrait pas affecter négativement le résultat de notre programme. Cependant, lorsque cela se produit sur tous les chemins de la logique, il s'agit sûrement d'un bug. Cette règle pose un problème lorsqu'une méthode contient plusieurs instructions de retour qui renvoient toutes la même valeur. La figure suivante illustre le cas mentionné ainsi que le tableau après la figure contient les attentions pour ce cas.



```
int foo(int a) {
  int b = 12;
  if (a == 1) {
    return b;
  }
  return b;  // Noncompliant
}
```

Figure 1 : Cas 1

Attention	context
0.194439	METHOD_NAME,(NameExpr1)^(MethodDeclaration)_(BlockStmt)_ (ReturnStmt)_(NameExpr0),b
0.124241	int,(PrimitiveType0)^(MethodDeclaration)_(NameExpr1),METHOD_NAME
0.112174	12,(IntegerLiteralExpr1)^(VariableDeclarator)^(VariableDeclarationExpr)^(ExpressionStmt)^(BlockStmt)_(IfStmt)_(BlockStmt)_(ReturnStmt)_(NameExpr0),b
0.111843	int,(PrimitiveType0)^(MethodDeclaration)_(Parameter)_(VariableDeclaratorId0),a
0.081677	a,(VariableDeclaratorId0)^(Parameter)^(MethodDeclaration)_(BlockStmt)_(ReturnStmt)_(NameExpr0),b

0.054237	12,(IntegerLiteralExpr1)^(VariableDeclarator)^(VariableDeclarationExpr)^(ExpressionStmt)^(BlockStmt)_(ReturnStmt)_(NameExpr0),b
0.048277	int,(PrimitiveType1)^(Parameter)^(MethodDeclaration)_(BlockStmt)_(IfStmt)_(BlockStmt)_(ReturnStmt)_(NameExpr0),b
0.047737	int,(PrimitiveType1)^(Parameter)^(MethodDeclaration)_(BlockStmt)_(ReturnStmt)_(NameExpr0),b
0.040247	int,(PrimitiveType0)^(MethodDeclaration)_(Parameter)_(PrimitiveType1),int
0.038881	b,(NameExpr0)^(ReturnStmt)^(BlockStmt)^(IfStmt)^(BlockStmt)_(ReturnStmt)_(NameExpr0),b

Tableau 1 : Attention du cas 1

2.2 Cas 2 :

Parfois les programmeurs implémentent dans des méthodes des expressions booléennes qui ne changent pas l'évaluation des conditions ce qui entraîne l'inutilité de ces dernières. La figure au-dessous montre le cas cité ainsi le tableau illustre les attentions de ce cas.

```
Void test(){
a = true;
if (a) { // Noncompliant
  doSomething();
}

if (b && a) { // Noncompliant; "a" is always "true"
  doSomething();
}

if (c || !a) { // Noncompliant; "!a" is always "false"
  doSomething();
}
}
```

Figure 2 : Cas 2

Attention	Context
0.151889	a,(NameExpr0)^(UnaryExpr:not)^(BinaryExpr:or)^(IfStmt)_(BlockStmt)_(ExpressionStmt)_(MethodCallExpr0)_(NameExpr1),dosomething
0.080400	b,(NameExpr0)^(BinaryExpr:and)^(IfStmt)_(BlockStmt)_(ExpressionStmt)_(MethodCallExpr0)_(NameExpr1),dosomething
0.069045	a,(NameExpr0)^(IfStmt)^(BlockStmt)_(IfStmt)_(BinaryExpr:or)_(UnaryExpr:not)_(NameExpr0),a

0.068842	a,(NameExpr1)^(BinaryExpr:and)^(IfStmt)^(BlockStmt)_(IfStmt)_(BlockStmt)_(ExpressionStmt)_(MethodCallExpr0)_(NameExpr1),dosomething
0.053182	c,(NameExpr0)^(BinaryExpr:or)^(IfStmt)_(BlockStmt)_(ExpressionStmt)_(MethodCallExpr0)_(NameExpr1),dosomething
0.049849	void,(VoidType0)^(MethodDeclaration)_(BlockStmt)_(IfStmt)_(BlockStmt)_(ExpressionStmt)_(MethodCallExpr0)_(NameExpr1),dosomething
0.049511	b,(NameExpr0)^(BinaryExpr:and)^(IfStmt)^(BlockStmt)_(IfStmt)_(BlockStmt)_(ExpressionStmt)_(MethodCallExpr0)_(NameExpr1),dosomething
0.048783	a,(NameExpr0)^(IfStmt)^(BlockStmt)_(IfStmt)_(BlockStmt)_(ExpressionStmt)_(MethodCallExpr0)_(NameExpr1),dosomething
0.026986	dosomething,(NameExpr1)^(MethodCallExpr)^(ExpressionStmt)^(BlockStmt)^(IfStmt)^(BlockStmt)_(IfStmt)_(BinaryExpr:and)_(NameExpr1),a
0.024368	void,(VoidType0)^(MethodDeclaration)_(NameExpr1),METHOD_NAME

Tableau 2 : Attention cas2

2.3 Cas 3 :

Une méthode peut couvrir plusieurs expressions booléennes ce qui entraîne un mauvais partitionnement logique en code d'où le SRP (Single Responsabilité Principale) (Principe SOLID) ne sera pas respecté. La figure suivante représente ce cas ainsi que le tableau au-dessous montre les attentions calculées.

```
public int computeCost(String drink, boolean student, int amount) {

        if (amount > 2 && (drink == GT || drink == BACARDI_SPECIAL)) {
            throw new RuntimeException("Too many drinks, max 2.");
        }
        int price;
        if (drink.equals(ONE_BEER)) {
            price = 74;
        }
        else if (drink.equals(ONE_CIDER)) {
            price = 103;
        }
        else if (drink.equals(A_PROPER_CIDER)) price = 110;
        else if (drink.equals(GT)) {
            price = ingredient6() + ingredient5() + ingredient4();
        }
        else if (drink.equals(BACARDI_SPECIAL)) {
            price = ingredient6()/2 + ingredient1() + ingredient2() + ingredient3();
        }
        else {
            throw new RuntimeException("No such drink exists");
        }
        if (student && (drink == ONE_CIDER || drink == ONE_BEER || drink == A_PROPER_CIDER)) {
            price = price - price/10;
        }
        return price*amount;
    }
```

Figure 3 : Cas 3

Attention	Context
0.115429	drink,(NameExpr0)^(MethodCallExpr)^(IfStmt)_(IfStmt)_(IfStmt)_(ExpressionStmt)_(AssignExpr:assign0)_(IntegerLiteralExpr1),110
0.080412	amount,(NameExpr0)^(BinaryExpr:greater)^(BinaryExpr:and)_(EnclosedExpr)_(BinaryExpr:or)_(BinaryExpr:equals)_(NameExpr0),drink
0.048724	student,(VariableDeclaratorId0)^(Parameter)^(MethodDeclaration)_(BlockStmt)_(ReturnStmt)_(BinaryExpr:times)_(NameExpr0),price

0.041892	amount,(NameExpr0)^(BinaryExpr:greater)^(BinaryExpr:and)^(IfStmt)_(BlockStmt)_(ThrowStmt)_(ObjectCreationExpr)_(StringLiteralExpr1),toomanydrinksmax
0.029873	amount,(VariableDeclaratorId0)^(Parameter)^(MethodDeclaration)_(BlockStmt)_(ReturnStmt)_(BinaryExpr:times)_(NameExpr0),price
0.025783	student,(VariableDeclaratorId0)^(Parameter)^(MethodDeclaration)_(BlockStmt)_(IfStmt)_(IfStmt)_(IfStmt)_(MethodCallExpr0)_(NameExpr0),drink
0.025271	drink,(VariableDeclaratorId0)^(Parameter)^(MethodDeclaration)_(Parameter)_(VariableDeclaratorId0),student
0.018854	amount,(VariableDeclaratorId0)^(Parameter)^(MethodDeclaration)_(BlockStmt)_(IfStmt)_(BlockStmt)_(ExpressionStmt)_(AssignExpr:assign0)_(NameExpr0),price
0.018078	student,(VariableDeclaratorId0)^(Parameter)^(MethodDeclaration)_(BlockStmt)_(ReturnStmt)_(BinaryExpr:times)_(NameExpr1),amount

Tableau 3: Attention cas 3

2.4 Synthèse :

D'après les tableaux des attentions précédentes pour les cas choisis, nous remarquons que les poids dominants ont des baisses valeurs (0.194439, 0.151889, 0.115429 ~ 0.1). Cependant, nous ne pouvons pas conclure qu'une baisse valeur du poids dominant implique que le code étudié est un code semelle car cela nécessite une évaluation à long terme.

Par exemple, cet article (https://arxiv.org/pdf/2002.06392.pdf) présente une approche de refactoring en utilisant le code2vec. Les chercheurs proposent une approche pour recommander une méthode de refactorisations qui repose sur la représentation basée sur le chemin du code (ASTPaths). Celui-ci est utilisé pour former un classificateur d'apprentissage automatique. Une telle représentation permet de capturer la sémantique du code de telle manière que des morceaux de code similaires sont mappés sur des vecteurs similaires, et ayant de tels plongements qui permettent de former un classificateur. L'approche suggère soit de déplacer la méthode vers une classe sémantiquement similaire ou le laisser dans sa classe d'origine. Ils ont évalué l'approche sur deux jeux de données : le premier consiste de plusieurs projets open source avec des méthodes déplacées manuellement et le second consiste en des projets avec une odeur Feature Envy injectée automatiquement.