논문 2021-1-5 http://dx.doi.org/10.29056/jsav.2021.06.05

code2vec을 이용한 유사도 감정 도구의 성능 개선

엄태호*, 홍성문**, 양준혁**, 장효석*, 도경구*†

Enhancing the performance of code-clone detection tools using code2vec

Taeho Um*, Sung Moon Hong**, Joon Hyuk Yang**, Hyo Seok Jang*, Kyung-Goo Doh*†

요 약

소스코드 표절은 원본 자료의 출처를 분명히 밝히지 않고 자신의 것처럼 사용하는 행위를 말한다. 소스코드 표절로 인한 문제는 법적인 분쟁을 다투는 경우까지 다양한 문제를 일으킨다. 소스코드의 표절 여부는 일반적으로 비교 대상 소프트웨어 프로젝트 내의 각 소스코드를 전수 비교하여 유사도를 측정하여 결정한다. 전수 비교는 표절 가능성이 전혀 없는 코드도 비교 대상에 포함하기 때문에 그만큼의 시간을 헛되이 소모한다. 소스코드 표절로 의심되는 비교 쌍만 선별하여 비교할 수 있으면 그만큼 비교 횟수는 줄어들게 되어 탐지 도구의 실행 속도를 향상시킬 수 있을 뿐만 아니라, 표절 가능성이 높은 부분만을 대상으로 탐지의 정확도를 높이는데 집중할 수도 있다. 본 논문에서는 code2vec 이라는 기계학습 모델을 활용하여 코드 클론으로 의심되는 소스코드들을 미리 분류하여 비교 횟수를 줄임으로써 소스코드 표절 탐지의 성능을 개선할 수 있음을 보인다.

Abstract

Plagiarism refers to the act of using the original data as if it were one's own without revealing the source. The plagiarism of source code causes a variety of problems, including legal disputes. Plagiarism in software projects is usually determined by measuring similarity by comparing every pair of source code within two projects. However, blindly comparing every pair has been a huge computational burden, causing a major factor of not using tools of better accuracy. If we can only compare pairs that are probable to be clones, eliminating pairs that are impossible to be clones, we can concentrate more on improving the accuracy of detection. In this paper, we propose a method of selecting highly probable candidates of clone pairs by pre-classifying suspected source-codes using a machine-learning model called code2vec.

한글키워드 : 프로그램 유사도, 프로그램 표절, 머신 러닝, 코드클론, 코드 비교

keywords : program similarity, program plagiarism, machine learning, code clone, code comparison

* 한양대학교 소프트웨어학부
** 한양대학교 컴퓨터공학과
† 교신저자: 도경구(email: doh@hanyang.ac.kr)
접수일자: 2021.06.05. 심사완료: 2021.06.17.
게재확정: 2021.06.20.

1. 서 론

소프트웨어 소스코드에서 유사 혹은 동일한 부분을 코드 클론이라고 정의한다[1]. 소프트웨어

를 개발 또는 유지보수 하면서 유사한 혹은 같은 기능을 수행하는 코드를 복사하여 재사용함으로써 작업 효율을 높일 수 있다. 하지만 이러한 이점에도 불구하고 소프트웨어 공학에서 코드 클론의 존재를 해로운 것으로 인식하고 있다[2]. 대규모 소프트웨어를 개발하거나 유지 보수하는 과정에서 코드 클론 중 일부만 수정하여 소프트웨어 결함으로 이어지는 경우가 허다하기 때문이다. 따라서 소스코드에 생길 수 있는 코드 클론은 개발 또는 유지보수 과정에서 부단히 찾아서 리팩토링하여 제거하는 것이 소프트웨어 공학의 개발 기본 수칙이다.

그런데 코드 클론은 소프트웨어 유지 보수를 위한 리팩토링의 대상이 되기도 하지만, 소스코드의 무단 복제를 판단하는 잣대로 사용되기도 한다. 특히 소스코드의 복제, 수정, 배포가 용이해지면서 불법 복제로 인한 저작권 침해가 심각한 사회적 경제적 문제로 대두되고 있는 상황이며[3], 이에 대처하기 위한 소프트웨어 프로젝트 간의 복제 여부의 기술적 판정 요구가 증가하고 있다.

사람의 눈으로 일일이 검사하여 코드 클론을 찾아내는 작업은 검사자의 능력과 주어진 시간에 따라 결과가 차별적일 뿐만 아니라 비경제적이다. 따라서 대용량의 소스코드에서 최대한 신뢰할 수 있는 수준에서 감내할 수 있는 시간 안에 자동으로 찾아주는 도구가 필요하다. 코드 클론을 자동으로 찾아주는 기존의 도구들은 다양한 탐지 방법을 이용해 구현한다[4]. 소스코드의 문자열, 토큰을 기반으로 분석하거나, 문법 트리, 그래프를 기반으로 분석하는 방법 등이 있다. 문자열과 토큰기반은 코드 클론 탐지에 있어서 상대적으로 속도가 빠르나, 트리 기반 분석과 비교해 정확성이 떨어진다. 트리를 기반으로 분석하는 방법은 토큰 기반 분석에 비해 정확도가 높지만 탐지 속도에서는 비효율적인 모습을 보인다[5].

하지만 앞서 말한 도구들은 분석 방법과는 독립적으로 비교 대상이 되는 소스코드 쌍을 하나도 빠짐없이 전수 비교할 수밖에 없기 때문에 시간 비용이 높다. 두 개의 프로젝트에서 비교해야 하는 쌍이 m, n이라고 하면 완전 탐색을 진행할 경우 O(m x n)시간이 필요하다. 비교해야 하는 대상이 많아질수록 코드 클론 탐지 도구의 탐지 시간이 비교 개수의 곱의 비율로 증가하기 때문에 비교 쌍을 줄이는 연구가 필요하다. 본 논문에서는 소스코드 분석 전 기계학습 모델을 이용해서 코드 클론으로 의심되는 소스코드 쌍을 미리 분류함으로써 무의미한 비교를 사전에 제거하여 코드클론 탐지의 효율성을 높이는 방법을 제안한다.

2. 관련 연구

코드 클론을 탐지하는 방법에는 크게 문자열 기반, 토큰 기반, 트리 기반, 그래프 기반으로 분류할 수 있다[6]. 문자열 기반의 방법은 주석이나 개행문자를 모두 제거한 뒤, 문자열의 직접적인 비교를 통하여 소스코드의 유사도를 검사하는 방법이다. 문자열 비교 알고리즘을 통해 빠르게 소스코드의 클론 여부를 판별할 수 있지만, 구문 구조를 고려하지 않기 때문에 정확도가 떨어질 수밖에 없다. 토큰 기반의 분석 방법은 소스코드를 토큰의 나열로 변환한 뒤, 일차원 비교를 하여 코드 클론을 탐지한다[7]. 이는 문자열 기반의 방법과 마찬가지로 여전히 구문 구조를 고려하지 않기 때문에 정확도는 높아질 수 없다는 근본적인 한계가 있다. 트리 기반의 방법은 소스코드를 AST(Abstract Syntax Tree)로 변경한 뒤, 트리를 비교하여 소스코드의 유사도를 측정한다[8]. 이 방법 또한 코드의 구문구조를 재배치시키는 경우 명백한 클론을 놓칠 수 있다는 한계가 있

다. 그래프 기반 분석 방법은 소스코드를 PDG(Program Dependency Grap)로 변경하여 소스코드의 코드 클론 여부를 검사한다[9]. PDG 기반 클론 검출 방법은 데이터 흐름과 제어 흐름의 정보가 포함된 PDG를 이용하여 클론을 검출하는 방법이다. 이 방법은 대규모 시스템에 적용이 어렵고 클론 검출에 많은 시간이 걸린다는 단점이 있다.

개발자들은 소스코드를 재사용하기 위해 원형 소스코드를 복제한 다음, 클론으로 보이지 않으려고 원본의 실행 의미를 그대로 유지하면서 모양새가 다르게 보이도록 소스코드를 변경하기도 한다. 일반적으로 코드 클론은 변경의 정도에 따라 표 1과 같이 네 가지 유형으로 분류한다[10][11]. 네 가지 코드클론 유형에서 하위 유형은 상위 유형의 모든 변경 방법을 포함한다. 예를 들면 Type 3은 명령어의 변경뿐만 아니라 상위 유형인 Type 1의 클론 조건인 공백과 주석, 레이아웃의 변경 그리고 Type 2의 식별자, 상수, 타입 변경까지 모두 포함한다. 앞서 말한 네 가지 코드 클론 유형을 보면 Type 4로 갈수록 탐지하기가 어렵고 탐지하기 위한 시간도 많이 소모한다. 하위 유형이 상위 유형 모두를 포함하기 때문에 Type 4를 검출할 수 있으면 모든 유형을 검출 가능하다고 볼 수 있다.

표 1. 코드 클론 유형 분류
Table 1. Classification of Code Clone Types

Type	특징
1	공백, 주석, 레이아웃을 제외하고 동일
2	식별자, 상수, 타입을 제외하고 구조적, 외형적으로 동일
3	명령어의 변경, 삽입과 삭제 제외하고 동일
4	서로 다른 구문을 사용하지만, 기능은 동일

3. 소스코드 비교 쌍 생성 모델

두 개의 프로젝트 내에 비교해야 할 소스코드가 각각 m, n개가 있을 때, 비교해야할 쌍은 총 $m \times n$ 개 이다. 비교 대상을 무작정 모두 비교하는 대신, 코드 클론의 가능성이 없는 비교 쌍을 선별하여 골라내어 제거한 다음 가능성이 있는 비교 쌍만을 대상으로 클론을 탐지할 수 있다면, 이를 통하여 절약한 시간을 좀 더 정밀한 탐지를 하는데 투입할 수 있다. 본 논문에서 제안하는 방법은 code2vec[12]이라는 기계학습 모델을 이용하면 메소드의 기능을 추측하여 메소드의 이름 후보를 제시해주는데, 이를 이용하여 비슷한 이름 후보 군을 선별하여 비교 대상 메소드 쌍으로 묶어주는 것이다.

본 논문에서 제안하는 방법의 전체적인 구성은 그림 1과 같다. 소스코드를 입력으로 받아서 Spoon을 통해 메소드를 추출하고, 추출한 메소드를 code2vec으로 미리 생성한 모델에 입력으로 넣어 메소드 이름을 예측하고 예측한 정보와 입력으로 넣은 메소드 정보와 1:1로 매칭하는 집합을 만든다. 비교 대상 프로젝트도 같은 과정을 거쳐 데이터를 만들어 비교 후 예측값이 비슷하거나 같은 것들의 쌍을 내어준다.

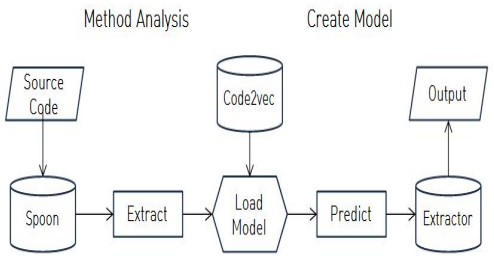


그림 1. 코드 클론 소스코드 쌍 생성 모델
Fig 1. Code clone source code pair generation model

3.1 code2vec

본 논문에서 사용하는 code2vec은 메소드 몸체 소스코드를 분석하여 기계학습으로 메소드의 이름을 제안해주는 도구이다. 즉 본 논문에서는 이 도구가 제안하는 메소드 이름이 유사하면 클론일 확률이 높다는 전제하에 코드 클론 탐지의 대상을 유사한 메소드 이름으로 한정하는 방법을 제안한다. code2vec은 코드 조각들을 연속 분포 벡터, 즉 코드 임베딩 형태로 표현하기 위해 제시된 신경망 모델이다. 코드를 입력받아서 AST로 변경해 분해하고 만들어진 트리에 대해서 특징들을 여러 개 묶어서 학습한다. 이런 학습 과정을 통해 의미적 유사성을 계산해 주어진 메소드의 기능을 예측해 적절한 이름을 반환한다[13].

그림 2는 f 메소드를 code2vec 모델에 적용하였을 때의 예시이다. code2vec 모델은 해당 메소드에 대해 sort, bubbleSort, reverse, reverseArray, heapify 등 가장 메소드의 이름으로 가능성이 높은 순으로 이름을 예측해주는 것을 확인할 수 있다. f라는 메소드의 이름만을 가지고는 해당 함수가 어떤 기능을 수행하는지 유추하는 것은 불가능에 가깝다. code2vec 모델을 해당 메소드에 적용함으로써 정렬 기능을 수행하고 있음을 확인할 수 있으며 실제 f 메소드 안에는 정렬 코드가 구현되어 있음을 볼 수 있다.

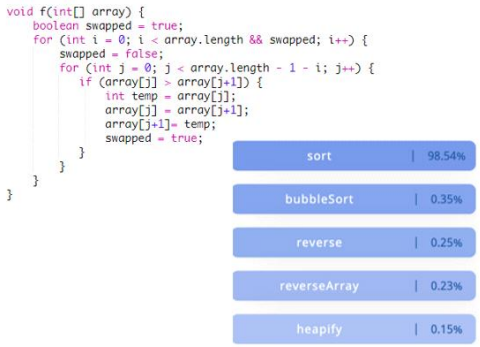


그림 2. code2vec 모델
Fig 2. code2vec model

3.2 Spoon

code2vec은 메소드의 몸체를 입력으로 받아 예측 결과를 내어준다. Java 소스코드 파일에서 code2vec을 이용하기 위해서는 소스코드 내에 존재하는 메소드 몸체의 추출이 필요하다. 이때, 메소드 이름이 오버로딩과 오버라이딩으로 인하여 메소드 이름이 동일한 경우가 생기며, 같은 함수가 서로 다른 파일 내에 존재할 수 있다. 따라서 메소드 추출뿐만 아니라 추후에 필요한 순서쌍을 주기 위해 메소드와 파일 안에서의 메소드 위치 정보, 파일 경로 등 해당 메소드에 대한 유일한 정보들이 필요하다. 이 작업을 수행하기 위해서 오픈소스 도구인 Spoon을 활용하였다. Spoon은 Java 개발자가 쉽고 간결한 방법으로 광범위한 도메인별 분석을 할 수 있도록 만들어진, 자바 소스코드를 분석, 재작성, 변환, 전송할 수 있는 도구로서, OW2 오픈소스 컨소시엄의 구성원이자 공식 인리아 오픈소스 프로젝트이다[14]. Spoon은 Java 소스코드 파일의 구문 분석을 하고 변환 API를 사용해 구축한 추상 구문 트리(Abstract syntax Tree)를 대상으로 정적 프로그램 분석을 시행한다.

3.3 모델

본 연구의 목표는 여러 가지 자바 프로젝트 내에 존재하는 메소드들에 대해 code2vec 모델을 적용하여 예측 결과를 도출해 코드 클론으로 의심되는 소스코드 쌍을 골라내는 것이다. 이 모델을 활용하면 그림 3과 같이 구문 구조는 다르지만 기능적으로 유사한 코드까지도 유력한 코드 클론의 후보로 찾아낼 수 있다. 사전에 GitHub 상의 9,500개의 주요 Java 프로젝트로 이루어진 Java-large 데이터 세트[15]로 학습한 code2vec 모델에 그림 3의 두 소스코드를 적용하면 두 코드 모두 delete, delete|dir를 포함하는 결과를 얻을 수 있다. 의미를 추론하는 code2vec의 특성상

모델을 통해 얻은 결과가 동일한 경우 기능적으로 유사한 코드 클론일 확률이 높다고 할 수 있다.

번호	소스 코드
1	private static void deleteDirectory(File f) {
2	File[] files = f.listFiles();
3	for (int i = 0; i < files.length; i ++) {
4	if (files[i].isDirectory()) deleteDirectory(files[i]);
5	files[i].delete();
6	}
7	}

번호	소스 코드
1	private static void deleteDir(File dir) {
2	File[] files = dir.listFiles();
3	for (File file : files) {
4	if (file.isDirectory()) deleteDir(file);
5	if (!file.delete()) {
6	System.err.println("ErrorMsg");
7	}
8	}
9	}

그림 3. 구문 구조가 다르지만 기능이 같은 소스코드
Fig 3. Source code with different syntax structures but the same functionality.

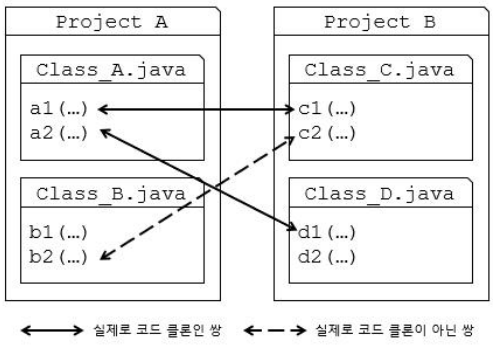


그림 4. 모델 적용 계획
Fig 4. Model Application Plan

그림 4와 같이 두 개의 자바 프로젝트가 주어지고, a1 메소드와 c1 메소드, a2 메소드와 d2 메소드가 코드 클론이라고 가정한다면, 가장 이상적인 상황은 정확히 두 번만의 비교로 두 소스코드 클론 쌍을 잡아내는 것이다. 모델을 적용하면 코드 클론으로 의심될 수 있는 소스코드 쌍은 [a1, d1], [a2, d1], [b2, c2] 등으로 나타난다. 만약 그림 4와 같은 상황에서 모든 쌍을 비교 대상으로 하여 코드 클론 탐색을 진행하게 될 경우 16회의 비교가 필요하게 된다. 그 대신에 앞서 서술한 세 개의 소스 코드 쌍에 대해서만 비교를 진행할 경우, [b2, c2]는 실제 소스코드 클론이 아니었기에 불필요한 비교가 되겠지만 완전 탐색을 진행할 때와 비교한다면 16회 대신 3회의 비교만으로도 코드 클론을 특정할 수 있기 때문에 비교 횟수와 시간적 측면에서 상당한 이익을 얻게 된다.

4. 실험 및 분석

4.1 데이터 집합

빅클론벤치[16]는 코드 클론 쌍에 대해 프로젝트의 이름, Java 파일의 이름, 코드 클론 타입, 함수의 시작 줄 번호, 함수의 시작 끝 번호, 각 코드 간의 유사도에 대한 정보를 담고 있다. 빅클론벤치의 모든 코드 클론 쌍에 대해 모델의 정확성을 검증하는 것은 시간상으로 어려움이 있다. 따라서 본 연구에서는 빅클론벤치 데이터 집합 내에 존재하는 코드 클론 쌍들을 유사도 순으로 정렬하고 이들 중 Type 3 또는 Type 4에 해당하는 70-30%의 유사도를 가진 클론 쌍을 50개를 선별하고 추가로 코드 클론과 연관성을 가지지 않는 자바 메소드를 담은 자바 파일 200개를 가져온 뒤, 절반씩 두 개의 데이터 집합으로 나누어 배치하였다.

표 2. 실험 데이터 세트
Table 2. Experimental Datasets

	Clone 개수	그 외
데이터 세트 1	50	100
데이터 세트 2	50	100

4.2 실험

code2vec은 메소드의 기능을 유추해 가장 가능성이 높은 순서대로 메소드의 적절한 이름을 내어준다. 데이터 집합 내에 존재하는 모든 소스코드들에 대해 생성한 모델을 적용해 나온 결과 중 상위 3개의 결과를 비교하여 동일한 예측 결과가 존재한다면 두 소스코드가 코드 클론일 가능성이 존재한다고 가정하였다.

그림 5에서 보이는 것과 같이 두 메소드는 모델을 적용하였을 때의 결과가 contains, fetch, exists로 상위 3개의 결과가 모두 동일하므로 코드 클론일 가능성이 높다고 판단하여 두 자바 메소드는 비교해야 할 소스코드 쌍에 추가하게 된다.

```
{
  "Path": "data1/clone1.java",
  "Package": "pack1",
  "Class": "A",
  "Method": "checkAppId",
  "Parameter": "[java.lang.String appid]",
  "Predicted": "contains,fetch,exists"
}
```

```
{
  "Path": "data2/cloneA.java",
  "Package": "pack2",
  "Class": "B",
  "Method": "verifyAppId",
  "Parameter": "[java.lang.String appid]",
  "Predicted": "contains,fetch,exists"
}
```

그림 5. 모델 적용 예시
Fig 5. Examples of model applications

각 데이터 집합마다 150개의 메소드를 담고 있어서, 완전 탐색을 진행하였을 경우 총 150 x 150 = 22,500회의 비교 횟수가 필요하다. 모델을 적용하여 코드 클론으로 의심되는 소스코드 쌍끼리만 비교를 진행하게 되면 비교 횟수를 비약적으로 줄일 수 있게 되며, 모델을 적용하여 얻어낸 소스코드 쌍에 포함된 실제 코드 클론의 개수가 모델의 성능을 검증하는 척도가 된다.

표 2의 데이터 집합에 대해 모델을 적용한 뒤, 전체 비교 횟수와 모델을 적용하여 의심되는 코드 클론 쌍들만을 비교했을 때의 비교 대상 메소드 쌍의 개수는 표 3과 같다.

표 3. 비교 대상 메소드 쌍의 개수의 변화
Table 3. Changes in the number of comparisons

완전 탐색 비교 쌍	모델 적용 비교 쌍
22,500	342

완전 탐색을 진행했을 때와 비교해 98.5%가량 비교 횟수가 감소한 것을 확인할 수 있다. Type 3 - 4 클론을 탐지하는 데에 있어 많은 시간적 비용이 소모됨을 고려했을 때, 총 342가지 비교 쌍 중 대부분의 코드 클론이 포함된다면 유의미한 결과를 얻어낼 수 있다. 데이터 집합에 포함된 코드 클론 쌍 중, 모델을 통해 얻어낸 342가지 비교 쌍에 속한 개수는 표 4와 같다.

표 4. 실험 결과
Table 4. Experimental Results

총 코드 클론	포함된 코드 클론	비율
169	161	95.3%

실험 데이터 집합에 총 50개의 코드 클론 쌍을 포함했지만 1:1 매칭이 아닌 1:n 매칭으로 코드 클론이 추가로 존재할 수 있음을 염두에 두어 삽입한 코드 클론 소스코드들을 눈으로 직접 검증하여 코드 클론의 개수를 산출한 결과 데이터 집합에는 총 169개의 코드 클론 쌍이 존재함을 확인했다.

표 4는 코드 클론을 탐지하는 데에 있어 모델을 적용한다면 1.5%의 비교만으로도 95% 이상의 코드 클론을 포함할 수 있음을 의미한다. 파일의 크기 및 소스코드의 라인 개수에 따라 모델을 적용하고 소스코드의 쌍을 추려내는 데에 얼마나 많은 시간이 소요되는지 측정한 결과는 표 5와 같다.

표 5. 모델 적용 소요 시간
Table 5. Model Application Time

구분	라인수	파일크기	시간
데이터 세트 1	2,498	84.16KB	128초
데이터 세트 2	2,969	104.14KB	

4.3 결과 분석

비교 횟수가 급격히 감소함에 따라 데이터 집합에 배치한 총 169개의 코드 클론 쌍 중, 모델이 코드 클론일 가능성이 높다고 판단한 코드 클론 쌍은 총 161개 존재했으며 약 5%의 코드 클론을 포함하지 못했음을 확인할 수 있다. 포함되지 않은 코드 클론들의 소스코드를 맨눈으로 직접 확인한 뒤, 변수 및 함수의 이름을 변경하거나 모델 적용 결과를 추가로 확인하는 등 여러 가지 시도를 거쳐 미탐 원인을 분석하였다. 발견된 미탐 유형은 다음과 같다.

유형 1. code2vec의 예측 결과 적용 문제

모델을 적용하였을 때 발생한 예측 결과 중 상위 3개의 예측 결과를 비교하여 동일한 부분이 존재했을 때 코드 클론일 가능성이 높다고 판단하였다. 그러나 <유형 1>은 빅클론벤치에서 클론이라고 정의한 소스코드 클론 쌍에서 상위 3개의 예측 결과까지는 동일한 결과를 보이지 않았으나, 하위 예측 결과에서 동일한 부분을 보인 경우이다. <유형 1>이 발생한 사례는 그림 6에서 확인할 수 있다.

두 코드는 빅클론벤치 데이터 집합에서 클론으로 정의되어 있으며 MD5 해시 함수를 이용해 유사한 기능을 수행하고 있기 때문에 모델을 적용하였을 때 코드 클론일 가능성이 높다고 판단하였어야 했으나 모델을 적용했을 때 나타난 소스코드 클론 쌍에 포함되지 못하였다. 두 코드의 모델 적용 결과는 그림 7과 같이 나타난다.

번호	소스 코드
1	static String getMD5Sum(String source) {
2	try {
3	MessageDigest digest = MessageDigest.getInstance("MD5");
4	digest.update(source.getBytes());
5	byte[] md5sum = digest.digest();
6	BigInteger bigInt = new BigInteger(1, md5sum);
7	return bigInt.toString(16);
8	} catch (NoSuchAlgorithmException e) {
9	throw new IllegalStateException("error");
10	}
11	}

번호	소스 코드
1	public static String getHashedPassword(String password) {
2	try {
3	MessageDigest digest = MessageDigest.getInstance("MD5");
4	digest.update(password.getBytes());
5	BigInteger hashedInt = new BigInteger(1, digest.digest());
6	return String.format("%1$032X", hashedInt);
7	} catch (NoSuchAlgorithmException nsae) {
8	System.err.println(nsae.getMessage());
9	}
10	return "";
11	}

그림 6. 미탐 유형 1 사례
Fig 6. Detection Failure Type 1 Case

```
1  "Path": "data1/clone32.java",
2  "Package": “pack1",
3  "Class": “c32",
4  "Method": “getMD5Sum",
5  "Parameter": "[java.lang.String source]",
6  "Predicted": “md,get,sum"
```

```
1  "Path": "data2/cloneC2.java",
2  "Package": “pack2",
3  "Class": “C32",
4  "Method": “getHashedPassword",
5  "Parameter": "[java.lang.String password]",
6  "Predicted": “hash,encrypt,digest"
```

그림 7. 그림 6의 모델 적용 결과
Fig 7. Model Application Results in Fig 6.

그림 7의 모델 적용 결과를 보면 3개의 예측값 모두 동일한 부분이 존재하지 않기 때문에 모델은 두 코드가 코드 클론이 가능성이 높지 않다고 판단하였다. 원인을 찾기 위해 모델의 소스코드를 일부 수정해 상위 3개의 예측값이 아닌 5개의 예측값을 내보내게끔 수정한 뒤 확인한 결과는 그림 8과 같다.

```
1 "Path": "data1/clone32.java",
2 "Package": “pack1",
3 "Class": “c32",
4 "Method": “getMD5Sum",
5 "Parameter": "[java.lang.String source]",
6 "Predicted": “md,get,sum,hash,password"
```

```
1 "Path": "data2/cloneC2.java",
2 "Package": “pack2",
3 "Class": “C32",
4 "Method": “getHashedPassword",
5 "Parameter": "[java.lang.String password]",
6 "Predicted": “hash,encrypt,digest,create,check"
```

그림 8. 예측값 추가 확인 결과
Fig 8. More Predictions Check Results

상위 3개의 예측값이 아닌 상위 5개의 예측값을 확인하였을 때, 결과에서 hash라는 동일한 부분이 발생했음을 확인할 수 있다. 모델을 적용할 시기에 상위 3개의 결과가 아닌 상위 5개의 결과를 토대로 코드 클론일 가능성이 높은 소스코드 쌍을 얻어냈다면 모델은 두 소스코드를 코드 클론일 가능성이 높다고 판단하여 비교해야 할 소스코드 쌍에 추가할 것이다. 이는 예측값을 조금 더 하위 단계까지 확인할 경우 요구되는 비교 횟수는 증가할 수 있으나 더 높은 정확도를 얻어낼 수 있음을 의미하므로 개선의 여지가 충분하다.

유형 2. 코드의 기능이 상이

두 번째 유형은 코드의 구문 구조에서 유사성을 보이나 수행하는 코드의 기능이 다른 경우이다. 이런 유형이 발생한 소스코드는 그림 9에서 확인할 수 있다. 그림 9 코드의 경우에는 코드의 구문 구조 측면에서 상당한 유사성을 보이는 것을 확인할 수 있기 때문에. 빅클론벤치에서는 마찬가지로 두 소스코드는 코드 클론이라고 명시되어 있다. 하지만 두 메소드의 3번 라인에서 SQL 문을 한쪽은 insert 구문, 다른 한쪽은 update 구문으로 정의되어 있어 서로 다른 기능을 하는 코드임을 확인할 수 있다.

제안하는 모델은 기능이 유사한 소스코드에 대해서 코드 클론일 가능성이 높은 메소드 쌍을 내어주기 때문에 이처럼 코드의 구문 구조에서 유사성을 보이나 기능에서 차이를 보이는 코드들에 대해서는 코드 클론일 가능성이 높지 않다고 판단하기 때문에 유형 2.와 같은 소스코드들의 경우 모델로 생성된 소스코드 비교 쌍에 포함되지 않아 기능이 유사 혹은 동일하지 않고 일부 구문 구조가 같다는 이유로 코드 클론으로 특정된 코드 클론들을 쌍으로 분류하기에 어려움이 있다.

번호	소스 코드
1	public boolean crear() {
2	int result = 0;
3	String sql =”insert into partida ---“;
4	try {
5	connection = conexionBD.getConnection();
6	connection.setAutoCommit(false);
7	ps = connection.prepareStatement(sql)
8	populatePreparedStatement(unaPartida);
9	result = ps.executeUpdate();
10	connection.commit();
11	} catch (SQLException ex) {
12	ex.printStackTrace();
13	try {
14	connection.rollback();
15	} catch (SQLException exe) {
16	exe.printStackTrace();
17	}
18	} finally {
19	conexionBD.close(ps);
20	conexionBD.close(connection);
21	}
22	return (result >0);
23	}

번호	소스 코드
1	public boolean update(int idPartida, partida partidaModificada) {
2	int intResult = 0;
3	String sql =“Update partida ---“;
4	try {
5	connection = conexionBD.getConnection();
6	connection.setAutoCommit(false);
7	ps = connection.prepareStatement(sql)
8	populatePreparedStatement2(unaPartida);
9	intResult = ps.executeUpdate();
10	connection.commit();
11	} catch (SQLException ex) {
12	ex.printStackTrace();
13	try {
14	connection.rollback();
15	} catch (SQLException exe) {
16	exe.printStackTrace();
17	}
18	} finally {
19	conexionBD.close(ps);
20	conexionBD.close(connection);
21	}
22	return (intResult >0);
23	}

그림 9. 미탐 유형 2 사례
Fig 9. Detection Failure Type 2 Case

5. 결론

본 논문은 code2vec 모델을 이용해서 메소드 단위로 기능이 유사할 것 같은 클론 쌍을 내주는 것을 제안하였다. 이를 이용하여 구문 구조가 다르더라도 기능이 유사한 코드들에 대해서 코드 클론을 탐색해야 할 상황이 발생할 경우 비교 횟수를 줄여 시간을 단축할 수 있을 것이다. 제안하는 방법으로는 메소드 안에 있는 작은 단위의 코드 클론을 찾을 순 없지만, 기능이 유사할 것 같은 메소드 단위를 찾는 코드 클론 도구에서는 유용할 것으로 생각된다. 기능이 유사할 것 같은 메소드를 찾는 기법은 큰 비용을 소모 하므로 제시한 방법을 같이 적용한다면 좋은 효과를 볼 수 있으리라 기대한다.

"본 연구는 과학기술정보통신부 및 정보통신기획평가원의 SW중심대학지원사업의 연구결과로 수행되었음"(2018-0-00192)

참 고 문 헌

[1] N.Saini et al., "Code Clones: Detection and Management", Proc. of the International Conference on Computational Intelligence and Data Science 2018. https://doi.org/10.1016/j.procs.2018.05.080

[2] 한상용, 박성윤, "메모리 액세스 로그 분석을 통한 프로그램 표절 검출", 정보처리학논문지 vol.13, no.6, pp.833-838, 2006.

[3] 손승우, 분쟁사례를 통해 본 DC 법률 문제 4. , 143(), 108-111, 2005.

[4] Chanchal K. Roy et al., "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach", Proc. of the Science of Computer Programming Volume 74, Issue 7, pp.470-495, 1 May 2009, https://doi.org/10.1016/j.scico.2009.02.007

[5] Pratiksha Gautam et al., "Various Code Clone Detection Techniques and Tools: A Comprehensive Survey", DOI: 10.1007/978-981-10-3433-6_79

[6] M.Sudhamani et al, "Code similarity detection through control statement and program features", 2019. https://doi.org/10.14801/jkiit.2020.18.3.79

[7] Hamid Abdul Basit et al., "Efficient token based clone detection with flexible tokenization", Proc. of the Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, 2007.

[8] 조영빈, 유철중, 이지현, "TBCNN 기반 코드 클론 유형 분류", 한국정보기술학회논문지, 제18권 제3호, 2020.

[9] 김연어, 이윤정, 우균, "클래스 구조 그래프 비교를 통한 프로그램 표절 검사 방법", 한국콘텐츠학회, JCCC 2013. http://dx.doi.org/10.5392/JKCA.2013.13.11.037

[10] 최성하, 도경구, "벨론 레퍼런스 코퍼스를 기준으로 exEyes의 재현율 평가", 한국소프트웨어감정평가학회 논문지, 제11권 제1호, 2015. http://www.i3.or.kr/html/paper/2015-1/(4)2015-1.pdf

[11] 박건우, 홍성문, "트리 기반 컨볼루션 신경망을 이용한 BigCloneBench 개선", 한국소프트웨어감정평가학회 논문지 제15권 제1호, 2019. http://dx.doi.org/10.29056/jsav.2019.06.05.

[12] Uri. Alon etal, "code2vec: learning distributed representations of code", Proc. of the ACM on Programming Languages Vol.3, 2019. https://doi.org/10.1145/3290353

[13] 양준혁, 모지환, 홍성문, 도경구, "code2vec 모델을 활용한 소스코드 보안 취약점 탐지", 한국소프트웨어감정평가학회 논문지 제16권 제2호, 2020. http://dx.doi.org/10.29056/jsav.2020.12.05

[14] Renaud Pawlak et al., “Spoon: A Library for Implementing Analyses and Transformations of Java Source Code”. Proc. of the Software Practice and Experience 2015. https://doi.org/10.1002/spe.2346
[15] code2vec, Java-large dataset https://github.com/tech-srl/code2vec
[16] BigCloneBench, https://github.com/clonebench/BigCloneBench, June 2017.

저 자 소 개

엄태호(Tae Ho Um)

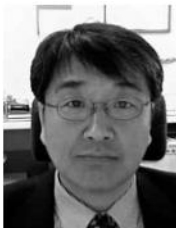
2017-현재 : 한양대학교 ERICA 소프트웨어학부 학사과정
<주관심분야> 프로그래밍 언어, 프로그램 분석

장효석(Hyo Seok Jang)

2020-현재 : 한양대학교 ERICA 소프트웨어학부 학사과정
<주관심분야> 프로그래밍 언어, 프로그램 분석

홍성문(Sung Moon Hong)

2012 위덕대학교 컴퓨터공학과 학사
2014 한양대학교 컴퓨터공학과 석사
2014-현재 : 한양대학교 대학원 컴퓨터공학과 박사과정
<주관심분야> 프로그래밍 언어, 프로그램 분석, 소프트웨어 보안



도경구(Kyung-goo Doh)

1980 한양대학교 산업공학과 학사
1987 아이오와주립대학 컴퓨터과학 석사
1992 캔자스주립대학 컴퓨터과학 박사
1993-1995 일본 아이주 대학 교수
1995-현재 : 한양대학교 ERICA 소프트웨어학부 교수
<주관심분야> 프로그래밍언어, 프로그램 분석, 소프트웨어 보안, 소프트웨어 공학

양준혁(Joon Hyuk Yang)

2020 한양대학교 ERICA 소프트웨어학부 학사
2020-현재 : 한양대학교 대학원 컴퓨터공학과 석사과정
<주관심분야> 프로그램 분석, 소프트웨어 보안