정 보 보 호 학 회 지
제 32 권 제5호, 2022. 10

프로젝트의 구조적 정보를 활용한 인공지능 기반 오픈소스 소프트웨어 식별

노 우 현*, 윤 종 원**

요 약

소프트웨어의 개발에서 오픈소스 소프트웨어(Open Source Software, 이하 OSS)의 사용이 급증하고 있다. 이에에 많은 OSS의 재사용 및 OSS간 재사용으로 인하여 OSS의 신규 취약점 대응 및 관리가 어려워지고 있다. 따라서 본 논문에서는 인공지능을 이용하여 OSS 프로젝트의 구조적 정보를 활용하여 여러 가지 OSS 재사용 기법에 대응하는 개선된 OSS 재사용 탐지(OSS Cloning Detection, OCD) 기술을 제안하고 그 성능을 평가한다.

Ⅰ. 서 론

최근 국내 기업의 60% 이상이 오픈소스 소프트웨어(Open Source Software, 이하 OSS)를 도입하여 사용하고 있을 정도로 OSS의 사용량은 꾸준히 증가하고 있다. 하지만 매년 4천 개 이상의 OSS 취약점이 신규로 발견되고 있어 안전한 소프트웨어의 개발과 운영을 위해 사용 중인 OSS를 지속해서 식별하고 취약점을 확인하는 관리의 필요성 역시 대두되고 있다.

OSS를 관리하고 사용 중인 OSS 중 취약점이 포함된 구성 요소를 확인하기 위해서는 입력으로 받은 프로젝트 소프트웨어를 빌드하고 생성하기 위해 사용되는 소스, 설정 파일 등 모든 파일들에 대해서 사용 중인 OSS의 정확한 식별이 선행되어야 한다.

본 논문에서는 인공지능으로 프로젝트의 구조적 정보를 학습하고 OSS를 정확하게 식별하는 OSS 재사용 탐지(OSS Cloning Detection, OCD) 기술을 제안한다.

Ⅱ. 코드 재사용 탐지 연구

외부 OSS를 개발 환경에 도입할 때에는 개발된 코드에 맞게 OSS를 수정[1]하여 사용하는 경우가 존재한다. 따라서 OSS의 사용을 확인하기 위해서는 수정된 코드로부터 OSS를 식별하는 것이 필요하지만 이 과정은 오류가 발생하기 쉽다[2]. 이를 극복하고 정확하게 수정된 OSS 사용을 탐지하기 위해 코드 재사용 탐지와 같이 다양한 연구들이 진행되어왔다.

코드를 그대로 또는 조금만 변경하여 사용하는 행위를 코드 재사용(Code Clone)[3]이라 한다. OSS를 사용하는 것도 코드 재사용의 일부이며 코드 재사용 탐지(Code Clone Detection, 이하 CCD) 기술을 이용하여 OSS 사용을 식별할 수 있다[4]. CCD 기술에서는 코드 재사용의 유형을 다음과 같이 크게 4가지로 나눈다:

1) 그대로 재사용(타입 1): 기존 코드를 변경 없이 그대로 재사용

2) 이름만 변경된 재사용(타입 2): 기존 코드에서 코드 구조는 그대로 둔 채, 변수 이름, 함수 이름 등만 변경하여 재사용

3) 거의 유사한 재사용(타입 3): 기존 코드에서 약간의 코드를 추가, 삭제하여 재사용

4) 의미적으로 동일한 재사용(타입 4): 기존 코드와 의미는 동일하지만 코드 구조는 전혀 다른 재사용

이 중 타입 4는 코드 구조가 전혀 달라서 관점에 따라 OSS를 재사용했다고 보기 힘들다. 따라서 OSS 식별 및 재사용 탐지를 위해서는 재사용 타입 4는 고려하지 않는다. 다음 표는 타입 1, 2, 3를 탐지하는 CCD 도구 별 성능을 보여준다.

본 연구는 중소벤처기업부의 기술개발사업 지원에 의한 연구임. [S2980557]
* 주식회사 스패로우 (책임연구원, whrho@sparrowfasoo.com)
** 주식회사 스패로우 (수석연구원, jwyoon@sparrowfasoo.com)

〔표 1〕 현재 제안/개발된 CCD 도구 정확도 비교

도구	기반 기술	정확도		재사용 타입
		정밀도	재현율	
SourcererCC[5]	토큰 기반	83%	90%	1,2,3
CCLearner[6]	토큰 기반	93%	N/A	1,2,3
CCAlinger[7]	토큰 기반	83%	92%	1,2,3
VUDDY[8]	본문 기반	100%	82%	1,2
CloneWorks[9]	메트릭 기반	93%	N/A	1,2,3
Vincent[10]	메트릭 기반	93%	92%	1,2,3

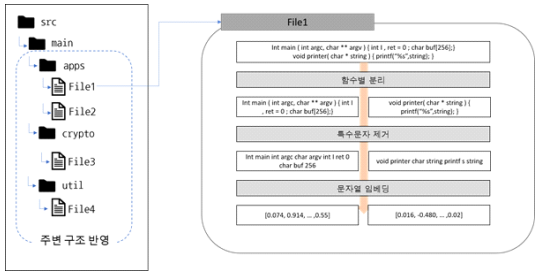
CCD는 코드 그 자체의 재사용을 탐지하는 것이 목적이므로, 해당 코드가 어디에 위치하는지는 고려하지 않는다. 그러나 OSS는 릴리즈 시에 코드의 내용뿐 아니라 코드의 위치 및 어떤 코드와 함께 있는지 등 프로젝트의 구조적 정보까지 정해지므로 정확한 OSS 재사용 탐지를 위해서는 프로젝트의 구조적 정보도 함께 고려되어야 한다.

본 논문에서 제안하는 프로젝트의 구조적 정보(주변 맥락)을 고려하여 코드 재사용(대상의 의미)을 탐지하는 기술은 자연어 처리 분야에서 문장에서 단어가 어디있는지, 어떤 단어와 같이 있는지 등(주변 맥락)을 고려하여 단어의 의미(대상의 의미)를 파악[11][12]하는 기술과 유사하다. 전통적으로는 문맥을 고려하기 위해 n-grams[13] 등으로 어휘적 특성을 반영한 뒤 선형 모델 또는 커널 모델을 이용하였지만, 최근에는 인공지능 기반의 접근법도 연구되고 있으며 전통적인 방법 대비 더 효율적[14]로 대상의 의미를 파악할 수 있다.

Ⅲ. 제안하는 방법

제안하는 식별 모델은 OSS 프로젝트의 구조 정보와 파일의 내용을 반영하여 OSS 재사용을 식별한다. 본 모델은 재사용 3번 타입까지 지원하며 기존 CCD 기술을 활용한 방법에 비해 높은 정밀도와 재현율을 목표로 한다. 특히 OSS 특징 정보들을 추상화하지 않고 모두 학습하기에는 학습 데이터가 OSS가 수집됨에 따라 무한하게 증가하는 문제가 존재하는데[15]. 이를 해결하기 위해 제안하는 모델은 문자열 임베딩 기술[16]을 사용하여 특징 정보의 손실을 최소화하며 효율적으로 학습에 반영한다.

파일의 계층 구조와 파일의 내용을 임베딩하는 계층(파일 임베딩 계층)은 텍스트 유사도 학습에서 널리 쓰이는 모델을 사용한다[17][18]. 파일 임베딩은 문자의 나열이 의미를 가질 수 있도록 사람이 의미를 이해할 수 있는(Human-readable) 파일(소스 코드, 텍스트 파일, 설정 파일) 등을 대상으로 한다. 입력 파일 주변에 같이 있는 파일들은 입력 파일과 동일하게 파일 임베딩 계층을 사용해 임베딩한다. 임베딩된 입력 파일과 주변 파일들을 입력으로 OSS를 임베딩하는 계층(OSS 임베딩 계층)은 문서를 식별할 때 널리 쓰이는 모델을 위주로 사용한다[19].



〔그림 1〕 제안하는 방법의 소스코드 임베딩 생성 과정

그림 1은 OSS 식별 모델 구조 및 모델을 통한 임베딩 생성 과정의 예시이다. 모델은 File2를 입력으로 받아 OSS를 식별하기 위해 주변 파일들(File1, File3, File4)과 계층 정보(src,main,apps,File1) 그리고 파일의 내용을 학습하여 파일의 OSS 정보를 임베딩한다. 이후 생성된 OSS 임베딩 벡터는 유사도를 측정하는 코사인 유사도 등을 통해 가장 유사한 OSS 임베딩 벡터를 DB에서 찾아 추천하는 데에 사용된다.

3.1. Softeare Embedding Model (SEM)

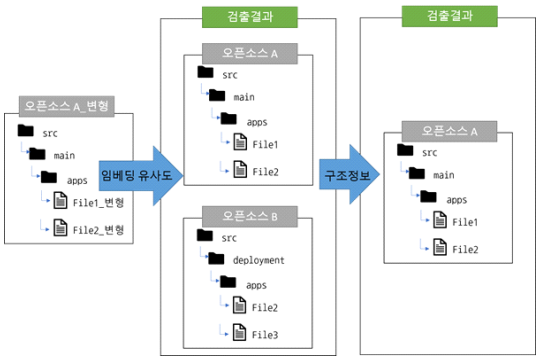
SEM 모델은 OSS를 임베딩(embedding) 하는 모델을 말한다. 인공지능을 위한 텍스트 전처리 분석 기법의 하나인 워드 임베딩은 현실 세계에 존재하는 자연어로 이루어진 단어들을 고정된 차원의 실수 벡터로 변환시키는 과정으로 분산 표현이라고도 한다. 이러한 벡터 표현은 다양한 응용이 가능한 자연어 처리 기반으로 활용된다. 본 표현법으로 각 소스코드를 각각의 512차원의 벡터로 표현 할 수 있는데, 본 연구에서는 다음과 같이 여러 가지 SEM 모델링 방법을 도출하여 비교하였다.

(방법 1) 소스코드 1본당 하나의 벡터를 산출하고, 하나의 OSS가 이루는 소스코드 전체의 벡터, 즉 OSS

가 표현하는 하나의 벡터를 벡터 평균 및 내적을 통해 계산한다. 각각의 OSS 식별에서 유사한 재사용 타입 2, 3을 지원하기 위해 변형된 오차 중 일부를 반영하는 허용치(threshold)를 반복 실험을 통해 확보하고 각 벡터의 코사인 유사도가 해당 오차 이내일 때 이 둘의 OSS는 일치한다고 계산한다.

(방법 2) 소스코드 1본당 하나의 벡터를 산출 할 때, 소스코드의 원본을 이용하는 것이 아닌 의미없는 특수기호 및 공백의 제거를 통해 하나의 문장 형태로 소스코드를 압축하고 그것에서의 각 소스 1본당 벡터를 산출하여 각각의 OSS 식별에서 유사한 재사용 타입 2, 3을 지원하기 위해 변형된 오차중 일부를 반영하는 허용치를 반복 실험을 통해 확보하고 각 벡터의 코사인 유사도가 해당 오차 이내일 때 이 둘의 OSS는 일치한다고 계산한다. 그리고 동시에 소스코드를 라인 단위로 분리하여 25개 라인 단위로 벡터를 생성한다. 각각의 OSS 식별에서 유사한 재사용 타입 2, 3을 지원하기 위해 각 벡터의 코사인 유사도가 사전에 정의된 해당 오차 이내일 때 이 둘의 OSS는 일치한다고 계산한다. 마지막으로 적절한 성능의 소스코드 해석기가 있을 때, 소스코드를 함수 단위로 분해하고, 각 함수별로 하나의 벡터를 산출한다. 이 때, 단순한 getter 및 setter 함수로 인해 발생할 수 있는 벡터의 노이즈를 제거하기 위해 함수 길이가 5라인 이하인 함수는 대상에서 제외한다.

(방법 3) 소스코드의 구조적 정보를 이용하는 방법으로 소스코드를 이루는 파일의 이름 및 주변 파일들의 이름 그리고, 디렉토리 구조를 트리 형태로 추상화하여 이 구조를 완벽히 포함하고 있는지를 집합연산으로 비교한다. 제안하는 방법 2에 구조 정보를 사용하는 방법을 조합하여 사용한다면 어떤 OSS 프로젝트



〔그림 2〕 제안하는 방법 3에 의한 OSS 식별 흐름

A를 프로젝트 B가 내부에 포함된 디렉토리에 복제하여 사용하는 형식으로 사용하는 경우, 방법 1, 2에서 발생할 수 있는 OSS 다중 검출 문제를 해소할 수 있다. 그림 2는 제안하는 방법 3에 의한 흐름을 도식화한 것이다.

Ⅳ. 실험 및 결과

4.1. 실험 대상

실험 대상으로는 수집된 OSS 중 1천개의 C/C++ OSS 들을 무작위로 선별하였다. 이중 헤더파일을 제외하고 .c .c++ .cc .cxx 파일만을 소스코드로 간주하여 실험을 진행하였다.

실험은 순차적으로 하나의 OSS를 선정하여 1,000개의 OSS에 대하여 제안하는 방법 대비 올바른 OSS를 검출하는지 여부 1,000건을 누산하였다. 실험에 선정된 1,000개의 OSS 를 대상으로 원본을 재사용한 경우와 유사한 재사용 (타입 2, 3) 경우에 대해서 재사용 여부를 탐지한 결과가 얼마나 일치하는지를 계산하고, 정확하게 일치하는 하나의 오픈소스만을 탐지한 경우를 그 일치로 산정하고 정확하게 일치하는 오픈소스 프로젝트를 검출하더라도 다른 오픈소스 또한 같이 탐지하였다면 이는 거짓-긍정(False-Positive)로 산정하였다.

4.2. 성능 지표 및 측정 방법

본 평가에서는 제안하는 방법의 효과성을 보이기 위해 구조 정보를 포함하지 않고 일치성만을 판단하는 방법(방법 1, 2)와 구조 정보를 추가 반영한 방법(방법 3)에 대해 모두 평가를 수행하여 제안하는 방법으로 구조 정보를 포함하여 일치성을 판단하는 경우 성능이 향상되는지를 함께 확인한다. 표 2는 본 논문에서 확인한 주요 성능 지표 및 측정 방법을 나타낸다.

〔표 2〕 주요 성능 지표 및 측정 방법

성능 지표	지표 정의	지표 수	측정 방법
원본 OSS 재사용 탐지 정밀도	OSS 이름을 알고 있는 수정 없는 OSS	1,000	본 식별 기술이 추측한 파일의 OSS 이름과 실제 OSS 이름이 일치하는지 확인한다. (정밀도는 정오탐 대비 정탐의 비율)

성능 지표	시료 정의	시료 수	측정 방법
원본 OSS 재사용 탐지 재현율	상동	1,000	'원본 OSS 재사용 탐지 정밀도 실험'과 시료와 수행 방법은 동일하다. (재현율은 정미탐 대비 정탐의 비율)
유사 OSS 재사용 탐지 정밀도 (타입 2)	OSS 이름을 알고 있는 수정된 OSS	1,000	시료들의 일부를 재사용 타입 2에 기반하여 무작위로 변경한다. 시료 안에 포함된 파일들을 대상으로 본 식별 기술이 추측한 파일의 OSS 이름이 변경되기 전 원본 OSS 이름과 일치하는지 확인한다.
유사 OSS 재사용 탐지 정밀도 (타입 3)	OSS 이름을 알고 있는 수정된 OSS	1,000	시료들의 일부를 재사용 타입 3에 기반하여 무작위로 변경한다. 시료 안에 포함된 파일들을 대상으로 본 식별 기술이 추측한 파일의 OSS 이름이 변경되기 전 원본 OSS 이름과 일치하는지 확인한다.
유사 OSS 재사용 탐지 재현율 (타입 2)	상동	1,000	'유사 OSS 식별 정밀도 실험 (타입 2)'와 시료와 수행 방법은 동일하다.
유사 OSS 재사용 탐지 재현율 (타입 3)	상동	1,000	'유사 OSS 식별 정밀도 실험 (타입 3)'과 시료와 수행 방법은 동일하다.

4.3. 실험 결과

(방법 1) 아래 표 3, 4, 5는 방법 1에 대한 실험 결과를 보여준다. 방법 1 은 소스코드 전체를 벡터화하는 과정을 거치기 때문에, 전체 파일 단위의 벡터가 일치해야 일치한다고 할 수 있다. 원본 재사용 탐지의 경우 완전 복제이기 때문에, 이러한 형태의 복제는 무리없이 잘 검출 할 수 있다. 다만 다른 오픈소스 프로젝트를 내포하고 있는 경우를 오검출하는 사례가 존재한다. 원본 재사용 탐지의 FN 이 발생한 경우는, 저장소가 C/C++ 소스 없이 header파일로만 이루어진 경우가 있었기 때문이었다.

유사한 재사용 타입 2, 3의 경우 소스코드 전문 길이가 짧아 사소한 변화 자체가 벡터값을 크게 변화시키는 경우 검출력 하락에 영향을 미쳤다고 분석할 수 있다. 그러나 오탐률 및 미검출률을 볼 때 실용적으로 사용하기엔 성능이 현저히 부족함을 알 수 있다.

〔표 3〕 방법 1 - 원본 OSS 재사용 탐지 실험 결과

허용치	95%	98%	100%
TP	750	858	991
FP	241	133	0
FN	9	9	9
정확도	0.750	0.858	0.991
정밀도	0.757	0.866	1.000
재현율	0.988	0.990	0.991
F-measure	0.857	0.924	0.995

〔표 4〕 방법 1 - 유사 OSS (타입 2) 재사용 탐지 실험 결과

허용치	95%	98%	100%
TP	800	774	0
FP	148	67	0
FN	52	226	1000
정확도	0.800	0.725	0.000
정밀도	0.844	0.920	-
재현율	0.939	0.774	0.000
F-measure	0.889	0.841	-

〔표 5〕 방법 1 - 유사 OSS (타입 3) 재사용 탐지 실험 결과

허용치	95%	98%	100%
TP	803	673	0
FP	141	63	0
FN	56	264	1000
정확도	0.803	0.673	0.000
정밀도	0.851	0.914	-
재현율	0.935	0.718	0.000
F-measure	0.891	0.805	-

(방법 2) 표 6,,7, 8은 방법 2에 대한 실험 결과를 보여준다. 방법 2를 사용 한 경우, 유사한 재사용 타입 2와 3에서 미약한 개선점이 있음을 확인할 수 있었다. 세그멘테이션 과정에서 변경된 부분과 원본 그대로 보존된 부분이 혼재되어 원본 부분의 정보를 통해 일치하는 OSS를 조금 더 검출 할수 있었다. 소스코드의 분할 벡터화를 통해 다른 OSS의 오검출이 높아지고, 목적 OSS를 내포하고있는 또다른 OSS를 검출하는

〔표 6〕 방법 2 - 원본 OSS 재사용 탐지 실험 결과

허용치	95%	98%	100%
TP	819	897	958
FP	172	94	33
FN	9	9	9
정확도	0.819	0.897	0.958
정밀도	0.826	0.905	0.967
재현율	0.989	0.990	0.991
F-measure	0.900	0.946	0.979

〔표 7〕 방법 2 - 유사 OSS (타입 2) 재사용 탐지 실험 결과

허용치	95%	98%	100%
TP	745	841	947
FP	221	144	42
FN	34	15	11
정확도	0.745	0.841	0.947
정밀도	0.771	0.854	0.958
재현율	0.956	0.982	0.989
F-measure	0.854	0.914	0.973

〔표 8〕 방법 2 - 유사 OSS (타입 3) 재사용 탐지 실험 결과

허용치	95%	98%	100%
TP	740	835	949
FP	224	151	39
FN	36	14	12
정확도	0.740	0.835	0.949
정밀도	0.768	0.847	0.961
재현율	0.954	0.984	0.988
F-measure	0.851	0.910	0.974

〔표 9〕 방법 3 - 원본 OSS 재사용 탐지 실험 결과

허용치	95%	98%	100%
TP	991	991	991
FP	0	0	0
FN	9	9	9
정확도	0.991	0.991	0.991
정밀도	1.000	1.000	1.000
재현율	0.991	0.991	0.991
F-measure	0.995	0.995	0.995

〔표 10〕 방법 3 - 유사 OSS (타입 2) 재사용 탐지 실험 결과

허용치	95%	98%	100%
TP	991	991	991
FP	0	0	0
FN	9	9	9
정확도	0.991	0.991	0.991
정밀도	1.000	1.000	1.000
재현율	0.991	0.991	0.991
F-measure	0.995	0.995	0.995

〔표 11〕 방법 3 - 유사 OSS (타입 3) 재사용 탐지 실험 결과

허용치	95%	98%	100%
TP	991	991	991
FP	0	0	0
FN	9	9	9
정확도	0.991	0.991	0.991
정밀도	0.991	0.991	1.000
재현율	1.000	1.000	0.991
F-measure	0.995	0.995	0.995

사례가 많아졌다.

(방법 3) 표 9, 10, 11은 방법 3에 대한 실험 결과를 보여준다. 방법 3에서는 방법2에서 1차로 걸러진 결과를 토대로 주변 정보를 통해 2차 가공을 하였다. 주변 정보는 파일구조 및 주변 디렉토리 구조 등을 포함하여 일치하는 경우에만 검출하는 방법으로 이 때의 경우 다른 OSS가 목적 OSS를 내포하고 있더라도 주변 정보 구조가 서로 달라 일치하지 않는 것으로 판단하여 FP를 의미있는 수준으로 제거하고, 정확도를 올림을 알 수 있다.

각 방법의 실험 결과를 비교한 결과, 표 *과 같이 제안하는 방법 3의 경우 원본 OSS 재사용 탐지, 유사 OSS (타입2, 3 모두) 재사용 탐지에 있어서 방법 1과 2보다 모든 허용치에 대해서 우수한 정확도를 보여주었다.

〔표 12〕 방법별 재사용 탐지 실험 정확도 비교 결과 (허용치 100% 기준)

방법	방법 1	방법 2	방법 3
원본 OSS 재사용 탐지	0.991	0.958	0.991
유사 OSS 재사용 탐지 (타입 2)	0.000	0.947	0.991
유사 OSS 재사용 탐지 (타입 3)	0.000	0.949	0.991

Ⅴ. 결 론

본 논문에서는 오픈소스 소프트웨어의 코드를 그대로, 혹은 일부 변경하여 동일한 형식으로 재사용하는 것을 탐지하기 위해 프로젝트의 구조적 정보를 활용하여 인공지능 기반으로 OSS를 식별하는 방법을 제안한다. 제안하는 방법은 이름 및 주변 파일들의 이름 그리고, 디렉토리 구조를 트리 형태로 추상화하여 식별에 사용함으로써 다중 검출 문제를 해결함과 동시에 높은 정확도로 OSS를 식별할 수 있다.

실험 결과 원본 OSS를 재사용한 것뿐만 아니라 유사한 OSS (타입 2 및 3)의 재사용에 대해서도 평균 99.1%의 정확도로 OSS를 식별할 수 있음을 확인하였다.

향후 OSS간의 상호 참조 및 OSS의 사용성은 꾸준히 증가할 것으로 예상되므로 OSS 간의 상호 중첩 재사용 또한 증가할 것이다. 중첩된 OSS의 재사용을 정확하게 검출하여 중첩 재사용된 OSS에서 발생한 취약점 이슈 등에 대한 관리 기법에 관한 선도적인 연구가 지속적으로 이루어져야 할 것이다.

참 고 문 헌

[1] N. Saini, *et al.*, “Code Clones: Detection and Management”, *Procedia Computer Science*, vol. 132, pp. 718-727, 2018.

[2] R. Duan, *et al.*, “Identifying Open-Source License Violation and 1-day Security Risk at Large Sacle”, *CCS’ 17*, 2017.

[3] C. K. Roy and J. R. Cordy, “A survey on software clone detection research.” *Queen’s School Comput., Tech. Rep.,* vol. 115, pp. 64-68, 2007.

[4] Q. U. Ain *et al.*, “A Systematic Review on Code Clone Detection”, *IEEE Access*, vol. 7, pp. 86121-86144, 2019.

[5] H. Sajnani *et al.*, "SourcererCC: Scaling code clone detection to big-code," *In Proc. IEEE/ACM 38th Int. Conf. Softw. Eng. (ICSE)*, pp. 1157-1168, 2016.

[6] L. Li, H. *et al.*, “Cclearner: A deep learning-based clone detection approach,” *In Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, pp. 249-260, 2017.

[7] P.Wang, *et al.*, "CCAligner: A token based large-gap clone detector,“ *In Proc. 40th Int. Conf. Softw. Eng.*, pp. 1066-1077, 2018.

[8] S. Kim, *et al.*, “VUDDY:A scalable approach for vulnerable code clone discovery,” in Proc. IEEE Symp. Secur. Privacy (SP), May 2017, pp. 595-614.

[9] M. Sudhamani and L. Rangarajan, "Cloneworks: A fast and flexible large-scale near-miss clone detection tool,“ *In Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. Companion (ICSE-C)*, pp. 177-179, 2017.

[10] C. Ragkhitwetsagul, *et al.*, “A picture is worth a thousand words: Code clone detection based on image similarity,” *In Proc. IEEE 12th Int. Workshop Softw. Clones (IWSC)*, pp. 44-50, 2018.

[11] T. Mikolov *et al.*, “Distributed Representations of Words and Phrases and their Compositionality”, *NIPS*, pp 3111-3119.

[12] T. Mikolov *et al.*, “Efficient Estimation of Word Representations in Vector Space”, *ICLR Workshop Papers*.

[13] T. Markov, “Example of a statistical investigation of the text of “Eugene Onegin” illustrating the dependence between samples in chain”, *Bulletin de l’Academie Imperiale des Sciences de St.-Petersbourg*, vol. 7, pp. 153-162.

[14] Z.Yang *et al.*, “Hierarchical Attention Networks for Document Classification”, *In Proc. of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics:*

Human Language Technologies, pp. 1480-1489, 2016.

[15] F. Zui, *et al.*, “Neural machine translation inspired binary code similarity comparison beyond function pairs,” *arXiv preprint arXiv:1808.04706*, 2018.

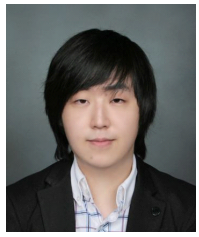
[16] Y. Kim, *et al.*, “Character-Aware Neural Language Models”, *arXiv preprint arXiv:1508.06615*, 2015.

[17] M. Seo *et al.*, “Bidirectional attention flow for machine comprehension”, In Proc. of ICLR, 2017.

[18] Devlin,J. *et al.*, “BERT: pre-training of deep bidirectional transformers for language understanding” *In Proc. of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, vol. 1, pp. 4171-4186, 2019.

[19] F. Hill *et al.*, “Learning distributed representations of sentences from unlabelled data.”, *In Proc. of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 1367-1377, 2016.

〈저자소개〉



노 우 현 (Rho Woohyun)

2013년 2월 : 건국대학교 인터넷미디어과 졸업
2015년 2월 : 연세대학교 컴퓨터과학과 석사
2015년 3월~2018년 5월 : 주식회사 파수 전임연구원
2018년 5월~현재 : 주식회사 스패로우 책임연구원
<관심분야> 애플리케이션 보안, 오픈소스 소프트웨어, 인공지능

윤 종 원 (Yoon Jongwon)

2009년 8월 : 연세대학교 컴퓨터과학과 졸업
2011년 8월 : 연세대학교 컴퓨터과학과 석사
2011년 6월~2018년 5월 : 주식회사 파수 선임연구원
2018년 5월~현재 : 주식회사 스패로우 수석연구원
<관심분야> 애플리케이션 보안, 오픈소스 소프트웨어, 인공지능