KIPS Trans. Softw. and Data Eng.
Vol.7, No.12 pp.461~468 pISSN: 2287-5905

https://doi.org/10.3745/KTSDE.2018.7.12.461

A Code Recommendation Method Using RNN Based on Interaction History

Heetae Cho† · Seonah Lee†† · Sungwon Kang†††

ABSTRACT

Developers spend a significant amount of time exploring and trying to understand source code to find a source location to modify. To reduce such time, existing studies have recommended the source location using statistical language model techniques. However, in these techniques, the recommendation does not occur if input data does not exactly match with learned data. In this paper, we propose a code location recommendation method using Recurrent Neural Networks and interaction histories, which does not have the above problem of the existing techniques. Our method achieved an average precision of 91% and an average recall of 71%, thereby reducing time for searching and exploring code more than the existing recommendation techniques.

Keywords : Software Engineering, Deep Learning, Interaction History

RNN을 이용한 동작기록 마이닝 기반의 추천 방법

조 희 태† · 이 선 아†† · 강 성 원†††

요 약

개발자들은 소프트웨어 개발과 유지보수 작업 중 하나의 코드를 수정하는데 들이는 시간보다 이를 위해 코드를 탐색하고 이해하는데 더 많은 시간을 소모한다. 코드를 탐색하는 시간을 줄이기 위하여 기존 연구들은 데이터 마이닝과 통계적 언어모델 기법을 이용하여 수정할 코드를 추천하여 왔다. 그러나 이 경우 모델의 학습 데이터와 입력되는 데이터가 정확하게 일치하지 않으면 추천이 발생하지 않는다. 이 논문에서 우리는 딥러닝의 기법 중 하나인 Recurrent Neural Networks에 동작기록을 학습시켜 기존 연구의 상기 문제점 없이 수정할 코드의 위치를 추천하는 방법을 제안한다. 제안 방법은 RNN과 동작기록을 활용한 추천 기법으로 평균 약 91%의 정확도와 71%의 재현율을 달성함으로써 기존의 추천방법보다 코드 탐색 시간을 더욱 줄일 수 있게 해 준다.

키워드 : 소프트웨어 공학, 딥러닝, 동작기록

1. 서 론

시간, 품질, 비용은 소프트웨어 프로젝트의 핵심 요소이다. 특히 소프트웨어 프로젝트의 많은 특징 중 한시성 때문에 시간은 품질과 비용, 그 외 여러 요소에 영향을 미친다[1]. 개발자들은 주로 소프트웨어를 개발하거나 유지보수 작업을 하는 동안 코드를 수정하는데 많은 시간을 소모한다. 특히 하나의 코드를 수정하는데 들이는 시간보다 코드를 수정하기 위해 관련된 코드를 탐색하고 이해하는데 더 많은 시간을 소모하게 된다[2].

※ 이 논문은 정부의 재원으로 한국연구재단의 지원을 받아 수행된 연구임 (No. NRF-2015R1C1A2A01055111, NRF-2018R1D1A1A02085551).
† 비 회 원 : 경상대학교 정보과학과 석사과정
†† 종신회원 : 경상대학교 항공우주및소프트웨어전공 조교수
††† 종신회원 : KAIST 전산학부 부교수
Manuscript Received : August 27, 2018
Accepted : November 8, 2018
* Corresponding Author : Seonah Lee(saleese@gnu.ac.kr)

이러한 시간의 소모를 줄이기 위하여 수정해야 할 코드를 어떻게 수정하면 되는지를 추천하는 많은 연구들이 진행되어 왔다. 예를 들면 통계적 언어모델 중 N-gram 모델을 사용해 코드의 출현 순서를 학습시켜, 코드가 누락되었을 확률을 계산할 수 있다[3]. 또한 누락된 코드를 추천하는 연구가 진행하였다[4]. 그 외에도 기존의 N-gram 모델이 코드의 순서와 일부 어휘정보만을 사용하기 때문에 발생하는 한계점을 해결하기 위해 코드의 의미를 이해할 수 있는 모델이 제시되기도 하였다[5]. 하지만 이러한 연구들 모두가 여전히 정확도가 낮다는 한계점을 가진다.

최근 딥러닝 기법이 소프트웨어 공학 문제에 우수한 성능을 보인다는 연구들이 많이 발표되었다. 그중에서 추천 문제에 딥러닝 기법을 사용한 연구들도 있었는데, 예를 들면 개발자가 하고자 하는 행위를 자연어로 입력하면 자동으로 Application Programming Interfaces (APIs)의 사용 순서를 추천 해주거나[6], Java언어의 APIs를 동일한 기능을 할 수 있는 C#언어

의 APIs로 변경을 추천해 주는 연구가 있었다[7]. 이러한 딥러닝 기법은 코드를 추천할 뿐만 아니라 학습하는 정보에 따라 다양한 추천을 할 수 있다. 예를 들면 버그와 개발자의 정보를 학습시켜, 새로운 버그의 해결을 위한 개발자를 추천하는 연구도 있었다[8].

우리는 개발자들의 동작기록과 딥러닝 모델을 사용하여 개발자들이 코드의 탐색과 이해를 거쳐 수정해야 하는 코드를 찾는 데까지 걸리는 시간을 줄이기 위한 연구를 진행한다. 본 연구는 동작기록 중 개발자가 탐색하거나 수정한 기록들만 추출해 시간순으로 나열하고, 특정한 문맥을 구성하고 딥러닝 모델에 학습시켜 수정할 파일을 추천하는 방법으로 진행한다.

우리의 모델은 실험 대상 프로젝트들에 대해 평균 약 91%의 정확도와 약 71%의 재현율을 달성하였으며 다음과 같은 공헌을 하는 연구결과로 볼 수 있다.

-1. 동작기록을 딥러닝 모델에 적용하기 적합함을 보인다.
-2. 동작기록과 딥러닝 모델을 사용하여 파일수준에서 수정이 필요한 위치를 추천할 수 있다.

본 논문의 구성은 다음과 같다. 먼저 2절에서는 우리의 연구를 진행하게된 동기를 설명하며, 3절에서는 동작기록과 RNN의 개념에 관해 설명한다. 다음으로 4절에서는 실험 계획을 설명하며, 5절에서는 실험 결과를 제시한다. 6절에서는 실험 결과의 질적 조사를 기술한다. 마지막으로 7절에서는 관련 연구를 제시하며, 8절에서는 우리의 연구에 대한 타당성의 위협을 제시한다. 9절에서는 결론을 기술하며 논문을 마무리 한다.

2. 연구 동기

우리는 실제 버그리포트와 동작기록을 조사하여 개발자들이 어떻게 수정하는 코드를 탐색하는지 조사했다. 먼저 이클립스의 Platform 프로젝트의 100799번 버그리포트는 특정 플러그인을 사용할 때 오류메시지 없이 종료된다는 내용을 보고하였다. 해당 버그리포트의 2008년 02월 18일의 개정기록을 보면 총 3개의 코드 파일에서 변경이 발생했다. 해당 버그리포트의 동작기록을 보면 총 7개의 파일에서 탐색 또는 수정이 발생했다. 탐색 및 수정은 총 56번이 발생하였으며, 그중 37번은 탐색이고, 19번은 수정이었다. 첫 번째 수정은 56번의 탐색 및 수정의 동작 중 6번째 동작에서 발생하였으며, 세 번째 동작이 발생하기 까지 5분이 걸렸고 첫 번째 수정이 발생하기 까지는 10분이 걸렸다. 모든 변경이 완료되기 까지는 약 2시간 30분이 걸렸다.

또 동일 프로젝트의 193832번 버그리포트는 특정 플러그인을 사용했을 때 뷰가 정상작동 하지 않는다는 내용을 보고하였다. 해당 버그리포트의 2007년 10월 29일의 개정기록을 보면 총 1개의 코드 파일에서 변경이 발생했다. 해당 버그리포트의 동작기록을 보면 16개의 파일에서 총 71번의 탐색 또는 수정이 발생하였다. 그중 58번은 탐색이고, 13번은 수정이었다. 첫 번째 수정은 71번의 탐색 및 수정의 동작 중 3번째 동작에서 발생하였으며, 두 번째 수정은 21번째 동작에서 발생하였다. 첫 번째 수정이 발생하기 까지는 1분도 채 걸리지 않았다. 하지만 첫 번째 수정의 발생부터 두 번째 수정이 발생하기 까지는 18번의 탐색으로 약 11분의 시간이 걸렸다. 모든 변경이 완료되기 까지는 약 23분이 걸렸다.

이처럼 소스코드 하나를 수정하기 위해서 개발자는 많게는 수십 배에 가까운 양의 연관된 코드를 탐색해야 하며, 이는 많은 시간을 소모한다. 우리의 연구인 동작기록을 사용한 코드의 위치 추천은 개발자가 해당 작업과 관련하여 수정해야 할 코드의 위치를 추천해 줌으로써, 해당 코드를 찾는 데까지 걸리는 탐색의 횟수와 시간을 줄일 수 있기를 기대한다

3. 동작기록과 Recurrent Neural Network

본 절에서는 실험에 사용되는 딥러닝과 Recurrent Neural Network (RNN)에 대해 설명하며, 동작기록을 딥러닝에 적용하는 방법을 기술한다.

3.1 딥러닝과 RNN의 개념

딥러닝은 사람의 뇌를 본뜬 인공 신경망을 바탕으로 데이터를 이용하여 학습시키는 방법이다. 학습된 인공 신경망 모델은 주로 새로운 입력 데이터에 대한 결과를 예측한다. 그중 Recurrent Neural Network(RNN)은 Convolutional Neural Network(CNN)과 함께 딥러닝의 대표적인 인공 신경망이다. RNN의 기본 동작은 Fig. 1과 같다.

Fig. 1의 X, Y, H는 각각 RNN모델에 대한 입력, 출력, 상태정보를 의미한다. 모델을 유닛수준으로 풀었을 때 각 유닛에 이전 유닛의 상태정보가 들어옴으로써 이전 입력들에 대한 정보를 유지하게 된다. 유닛수준의 각 파라미터들은 아래의 식으로 계산된다.

$$h_n = \tanh(W_x x_n + W_h h_{n-1}) \quad (1)$$

$$y_n = W_y h_n \quad (2)$$

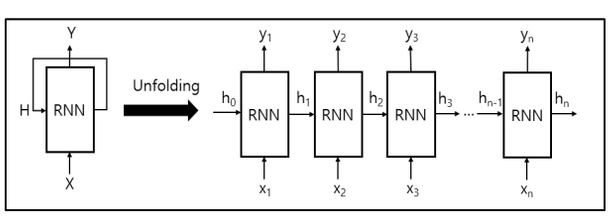

Fig. 1. Operation of Simple Recurrent Neural Network

위 식의 x_n, y_n, h_n은 각각 RNN유닛에 대한 입력 데이터, 출력 데이터, 상태정보를 나타낸다. 이 파라미터들은 각자의 가중치인 W_x, W_y, W_h와 곱해져 고유의 의미를 나타낸다. 학습은 상기의 식으로 계산하여 예측한 y값과 실제 y값의 차이를 이용해 손실함수를 계산하고, 최적화 함수를 통해 가중치

를 갱신함으로써 신경망을 학습시킨다.

다양한 연구가 진행되어 생긴 여러 가지 변형 RNN중에서 주로 사용되는 RNN은 long short-term memory (LSTM)[9]와 gated recurrent unit (GRU)이다[10]. 기본 RNN유닛과 LSTM, GRU의 차이점은 내부 구조에 있다. 기본 RNN유닛은 단순히 위의 Equation (1), (2)만으로 연산된다. 반면 LSTM과 GRU는 입력 게이트, 출력 게이트 등 기본 RNN유닛보다 복잡한 구조와 많은 연산이 필요하다. 그 대신 기본 RNN유닛보다 더 깊고 장기적인 학습이 가능하다

3.2 딥러닝에 동작기록 적용 방법

1) 동작기록

우리가 사용하는 동작기록은 Mylyn도구에서 생성하는 동작기록으로서 Fig. 2와 같이 xml 포맷으로 이루어져 있다. 이러한 동작기록은 단순히 수정된 코드의 정보를 기록한 개정기록과 달리 개발자가 코드를 탐색하거나 수정했던 모든 행위를 담는다. 이러한 동작기록은 개발자가 한 번의 코드 수정을 위해 코드를 어디서부터 탐색하기 시작했는지, 얼마나 많은 파일을 탐색했는지, 그리고 최종적으로 코드를 수정하는 데까지 얼마나 시간을 소비하였는지 등을 추론해 낼 수 있는 이점이 있다.

```
- <InteractionHistory Version="1" Id="https://bugs.eclipse.org/bugs-204358">
    <InteractionEvent StructureKind="java"
      StructureHandle="=org.eclipse.compare/compare&lt;org.eclipse.compare
      {CompareViewerPane.java" StartDate="2007-10-12 10:43:42.218 CEST"
      OriginId="org.eclipse.jdt.ui.CompilationUnitEditor" Navigation="null"
      Kind="selection" Interest="25.0" EndDate="2007-10-12 11:59:44.109 CEST"
      Delta="null"/>
    <InteractionEvent StructureKind="java"
      StructureHandle="=org.eclipse.compare/compare&lt;org.eclipse.compare"
      StartDate="2007-10-12 10:43:42.218 CEST"
      OriginId="org.eclipse.mylyn.core.model.interest.propagation"
      Navigation="org.eclipse.mylyn.core.model.edges.containment"
      Kind="propagation" Interest="154.69897" EndDate="2007-10-12 11:59:46.953
      CEST" Delta="null"/>
```

Fig. 2. Example of Interaction History

2) 동작기록 전처리

본 연구에서는 동작기록이 가진 많은 정보 가운데 개발자가 하나의 파일을 보거나 수정한 동작 행위와 그 행위를 시작한 시간과 끝난 시간, 그리고 해당 파일의 이름 정보를 사용한다. 이를 위해 우리는 먼저 python을 사용해 xml로 저장되어 있는 동작기록에서 우리가 사용할 정보들만 추출하였다. 다음으로 우리는 추출한 정보들을 모델에 학습시키기 위한 형태로 변형시켰다. 이 변형은 3개의 탐색 및 수정 파일과 1개의 수정 파일로 이루어지며 Fig. 3과 같다. Fig. 3에서 개발자가 순서대로 파일 A, B, C를 탐색한 다음 D, E를 수정하였을 때, ([A, B, C], D), ([B, C, D], E)로 탐색 및 수정한 세 개의 파일로 이루어진 집합과 수정한 하나의 파일로 정의되는 데이터 쌍 2개가 만들어 지며, Fig. 4에서 예를 볼 수 있다. Fig. 4에서 왼쪽은 Fig. 2의 동작기록에서 추출한 동작의 순서 중 일부이며, 동작은 위에서부터 아래로 진행되었다. 오른쪽의 두 박스는 추출한 동작을 모델에 학습시키기 위한 형태로 변형한 데이터 쌍의 모습으로, 각 박스의 점선 위 세 개의 파일은 탐색 및 수정된 파일의 집합으로 RNN모델의 입력 데이터에 해당하며, 점선 아래 하나의 파일은 위 세 번의 동작이후 수정된 파일로 입력 데이터의 정답에 해당한다.

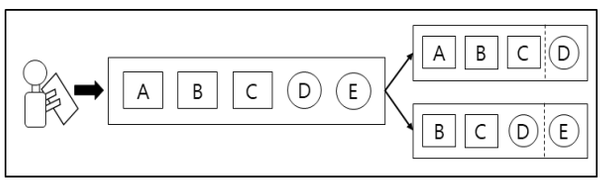

Fig. 3. Preprocessing of Interaction Histories

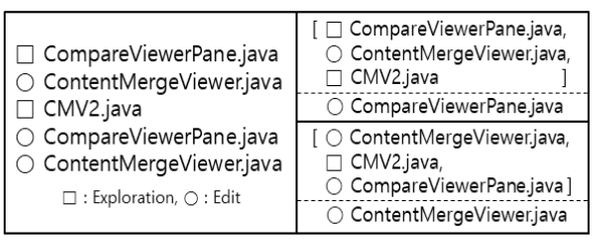

Fig. 4. Example of Data Pair

3) 모델 튜닝

우리의 모델은 Fig. 5와 같다. 학습 모델은 임베딩 계층과 RNN계층으로 구성된다. 임베딩 계층은 문자열인 파일 이름을 벡터로 바꾸기 위한 계층이다. RNN계층은 학습을 위한 계층으로 RNN계층을 통해 추천할 코드를 예측한다. 이때, 딥러닝 모델은 파라미터에 의해 성능이 변하기 때문에 우리는 먼저 적절한 파라미터를 찾기 위해 프로젝트 ECF를 대상으로 모델의 파라미터를 조율하였다. 대상 파라미터는 RNN계층의 유닛과 계층의 크기, 학습 횟수이며, 유닛은 [LSTM, GRU]로, 계층의 크기는 [100, 500, 1000]로, 학습 횟수는 [10, 100, 500, 1000]으로 진행하였다.

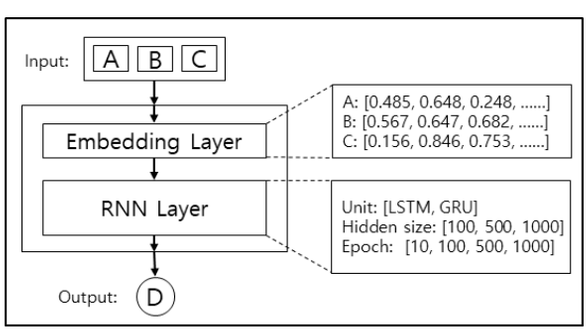

Fig. 5. Model Overview

4) 모델 학습

모델의 학습은 전처리된 동작기록의 데이터 쌍 중 랜덤으로 추출한 70%를 학습 데이터로 사용하였다. 학습의 손실함수는 PyTorch에 구현되어 있는 Equation (3)으로 계산되는 CrossEntropy를 사용하였으며, 최적화함수는 Adam[11]을 사용하였다.

$$CrossEntropy = -\frac{1}{N}\sum_{n=1}^{N}[y_n \log\hat{y} + (1-y_n)\log(1-\hat{y})] \quad (3)$$

Equation (3)에서 N은 모든 정답의 개수를 의미한다. y는 해당 입력 데이터에 대한 정답을 표현한 벡터로 정답일시 1을 오답일시 0의 값을 가지며, N의 크기를 가진다. y_n은 정답 벡터y의 n번째 값을 나타내며, $\hat{y}$은 해당 입력에 대해 모델이 예측한 값을 나타낸다. 이 식는 y_n이 1이면서 $\hat{y}$이 1에 근접하거나 y_n이 0이면서 $\hat{y}$이 0에 근접한다면 0에 가까운 수를 반환하며, y_n이 1이면서 $\hat{y}$이 0에 근접하거나 y_n이 0이면서 $\hat{y}$이 1에 근접한다면 매우 큰 수를 반환한다. 즉, 정답일 시에는 부담이 없지만, 틀렸을 시에는 큰 부담을 지게 하는 방법으로 점진적으로 해당 식의 값을 줄이는 방향으로 학습을 진행하게 된다.

5) 모델을 이용한 코드 추천 방식

우리의 RNN모델은 3개의 탐색 및 수정된 파일이 입력되면 1개의 수정 파일을 추천한다. 딥러닝 모델의 특성상 모든 입력에 대해 출력이 발생하는데, 이때 무분별하게 추천하는 결과를 방지하기 위해 Equation (4)로 계산되는 소프트맥스 함수를 사용해 예측의 정확률이 90%를 넘을 때만 추천하도록 임계값을 설정하였다.

$$Softmax(y_i) = \frac{e^{y_i}}{\sum_{i=1}^{m} e^{y_i}} \quad (4)$$

위 식에서 y_i는 입력에 대한 모델의 출력 중 i번째 벡터를 의미한다. e는 오일러 수이며, m은 y 벡터의 전체 개수를 의미한다. 소프트맥스 함수를 사용한 결과는 y값 전체의 합을 1로 만들어 준다.

4. 실험 계획

본 절에서는 실험을 통해 찾고자 하는 연구 질문과 실험의 대상 제시하고, 실험의 절차를 설명한다.

4.1 연구 질문

본 연구의 실험을 통해 아래의 질문들을 탐구한다.

RQ1. RNN모델을 사용한 추천의 성능은 어떠한가?

RQ2. RNN모델에 사용되는 파라미터들은 모델의 학습에 어떠한 영향을 미치는가?

RQ3. RNN모델과 기존 통계적 언어모델인 N-gram과의 차이는 무엇인가?

4.2 실험 대상

본 연구의 실험 대상은 장기적이고 다양한 정보를 가지고 있어야 하며, 일반적이어야 한다. 따라서 우리는 이클립스의 ECF, MDT, Mylyn, PDE, Platform, Others 프로젝트의 동작기록을 사용하기로 결정하였다. Others는 앞선 5개 프로젝트 이외의 여러 프로젝트의 동작기록을 가지고 있다.

Table 1은 각 프로젝트에 대한 전처리 이후 만들어진 데이터 쌍의 개수를 보여준다. 각 프로젝트 별로 누적된 동작기록의 수가 다르기 때문에 생성된 데이터 쌍의 개수에서 차이가 난다. 데이터 쌍이 가장 많은 프로젝트는 Mylyn이며 가장 적은 프로젝트는 PDE이였다. 평균 데이터 쌍의 개수는 약 36277개 이다. 각 프로젝트 별 전체 데이터 쌍의 개수 중 랜덤으로 추출한 70%를 모델의 학습 데이터로 사용하였으며, 나머지 30%는 학습된 모델의 테스트 데이터로 사용하였다.

Table 1. Number of Data After Preprocessing

Projects	ECF	MDT	Mylyn	PDE	Platform	Others
Number of training data	8425	17565	51471	4384	25359	45159
Number of test data	3612	7529	22059	1879	10869	19354
Total	12037	25094	73530	6263	36228	64513

4.3 실험 환경

실험의 모델은 Intel(R) Core(TM) i7-8700K 3.70GHz CPU와 16GB 메모리 환경에서 Python3.6과 딥러닝 라이브러리인 PyTorch를 사용해 구현하였다.

4.4 실험 절차

실험의 전체적인 진행은 먼저 동작기록을 딥러닝 모델에 학습 시킬 형태로 전처리를 진행한다. 다음으로 딥러닝 모델을 구현하고, 전처리된 동작기록을 학습시킨다. 마지막으로 딥러닝 모델의 성능을 평가한다.

4.5 척도

모델의 성능 평가를 위해 우리는 추천율, 정확도, 재현율을 측정하였으며 각 척도는 아래의 식과 같다.

$$\text{추천율} = \frac{\text{추천이 발생한 횟수}}{\text{추천을 시도한 횟수}} \quad (5)$$

$$\text{정확도} = \frac{\text{추천이 정답과 일치한 개수}}{\text{추천이 발생한 횟수}} \quad (6)$$

$$\text{재현율} = \frac{\text{추천된 수정 파일의 개수}}{\text{실제 수정파일의 개수}} \quad (7)$$

먼저 Equation (5)에서 추천율이란 추천을 시도 횟수 대비 추천이 실제 발생한 횟수를 의미한다. 우리 실험에서 추천 시도 횟수는 테스트 데이터의 개수로서 Table 1의 3번째 행에서 각 프로젝트의 테스트 데이터의 개수를 볼 수 있다. 추천

이 발생한 횟수는 테스트 데이터 쌍 중 우리가 설정한 임계값 90%를 넘어 추천이 발생한 개수이다. 다음으로 Equation (6)에서 정확도란 추천이 발생한 횟수 대비 추천이 정답과 일치한 개수를 의미한다. 우리의 실험에서 추천이 발생한 수는 추천율에서 설명한 것과 동일하며, 추천이 정답과 일치한 개수는 추천이 발생했을 때, 추천의 결과가 동작기록에서 입력된 3번의 동작 이후에 등장한 실제 수정된 파일과 일치하는 개수이다. 마지막으로 Equation (7)에서 재현율은 실제 수정파일의 개수 대비 추천된 수정파일의 개수를 의미한다. 우리의 실험에서 실제 수정파일의 개수는 동작기록에 존재하는 모든 수정파일의 개수이며, 추천된 파일의 개수는 모든 수정파일 중 우리의 모델이 추천한 수정파일의 개수이다.

5. 실험 결과

본 절에서는 4.1절의 각 연구질문에 답을 하며, 실험 결과를 보여준다.

5.1 RQ1 - RNN모델을 사용한 추천의 성능은 어떠한가?

우리는 먼저 모델의 성능 평가를 진행하기 위해 학습 데이터와 평가 데이터의 추출을 랜덤으로 5번 진행하였다. 이를 사용해 동일한 구조의 모델 5개를 생성하여 평가하였다. Table 2는 각 프로젝트에 대한 모델들의 평균값으로 성능을 평가한 결과표이다. 입력에 대한 예측 정확률의 임계값을 90%로 설정한 결과, 가장 높은 추천율을 보인 프로젝트는 ECF로 92%를 보였다. 가장 낮은 추천율을 보인 프로젝트는 Mylyn으로 59%를 보였으며, 평균 78%정도의 추천율을 보였다. 다음으로 정확도 또한 프로젝트 ECF가 96%로 가장 높았으며, 프로젝트 Mylyn이 84%로 가장 낮았다. 평균적으로 91%의 정확도를 보였다. 마지막으로 재현율은 프로젝트 ECF가 89%로 가장 높았으며 프로젝트 Mylyn이 50%로 가장 낮았다. 평균적으로 71%의 재현율을 보였다.

Table 2. Model Performance

Projects	ECF	MDT	Mylyn	PDE	Platform	Others
Amount	3612	7529	22059	1879	10869	19354
recommendation rate	0.92	0.78	0.59	0.84	0.76	0.77
precision	0.96	0.91	0.84	0.87	0.92	0.93
recall	0.89	0.71	0.50	0.74	0.70	0.72

5.2 RQ2 - RNN모델에 사용되는 파라미터들은 모델 학습에 어떠한 영향을 미치는가?

우리는 RNN모델의 구현을 위해 프로젝트 ECF를 대상으로 유닛, 계층의 크기, 학습 횟수를 대상으로 파라미터 조율을 진행했었다. Table 3은 각 파라미터를 조율했을 때 각 파라미터에 대한 모델 별 성능을 보여준다. 그 결과 상위 5개에서 유닛에는 모두 GRU가 등장했으며, 계층의 크기는 100과 500이 나타났다. 학습 횟수는 100, 500, 1000이다. 전체적으로는 먼저 유닛은 학습 횟수가 적을 때 GRU가 LSTM보다 성능이 조금 더 높지만, 그 외의 조건에서는 별다른 차이가 없었다. 다음으로 계층의 크기는 학습 횟수가 적을 때는 100보다 500, 1000의 성능이 약간 좋았지만, 이외에는 1000이 100, 500보다 약간 더 낮은 성능을 보였다. 마지막으로 학습 횟수는 10과 100사이에서 성능의 변화가 크지만 100이후에는 성능의 변화가 거의 없다. 우리는 성능과 학습 시간을 고려해 최종적으로 모델의 파라미터를 유닛은 GRU로, 계층의 크기는 500으로, 학습 횟수는 100으로 설정하였다.

Table 3. Result of Parameter Tuning

Unit	Hidden size	Epoch			
		10	100	500	1000
GRU	100	82.00	92.86	92.86	92.80
	500	86.60	92.88	92.72	92.66
	1000	86.07	92.41	92.36	92.36
LSTM	100	76.30	92.33	92.55	92.28
	500	83.94	92.05	92.03	92.03
	1000	83.67	91.81	91.69	91.67

5.3 RQ3 - RNN모델과 기존 통계적 언어모델인 N-gram과의 차이는 무엇인가?

우리는 RNN모델과 N-gram의 차이가 어느 부분에서 발생하는가에 대한 논의를 진행하였다. N-gram은 N개의 입력 데이터에 대한 정답 데이터를 학습하여 통계적으로 입력에 대한 정답을 예측하는 모델이다. N을 3으로 설정한 tri-gram 모델은 3개의 입력 데이터를 갖는다. 즉, 우리의 실험 중 동작기록의 전처리과정은 tri-gram과 동일하다. 따라서 우리는 모델 부분에 집중하여 논의를 진행한 끝에 RNN모델과 N-gram의 가장 큰 차이는 모델의 학습 방법과 결과의 출력 방법에 있다고 결론 내렸다. RNN모델은 데이터를 벡터화 하여 가중치와의 벡터 곱을 통해 고유의 가중치를 갱신하여 학습하기 때문에 입력되는 데이터의 벡터가 비슷하면 거의 동일한 결과를 출력한다. 하지만 기존의 N-gram은 가중치를 사용하지 않는 통계적 학습으로, 데이터를 벡터화 하더라도 입력되는 데이터가 학습된 데이터와 정확하게 일치하지 않으면 결과의 출력이 발생하지 않는 다는 점이다.

Fig. 6과 7은 동일한 학습 데이터를 사용하여 RNN과 tri-gram의 정확도와 재현율을 비교한 그래프이다. Fig. 5에서 RNN을 사용한 모델과 N-gram을 사용한 모델의 정확도를 비교했을 때 N-gram모델과 RNN모델 모두 최소 80% 이상의 정확도를 보인다. 하지만 Fig. 7에서 재현율을 비교했을 때 RNN을 사용한 모델이 N-gram모델에 비해 두 배 이상 우수함을 볼 수 있었다.

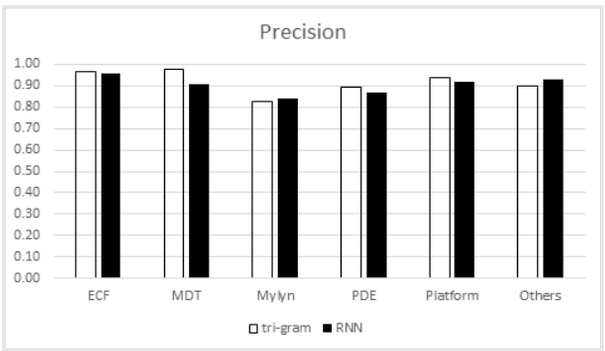

Fig. 6. Precision of N-gram and RNN

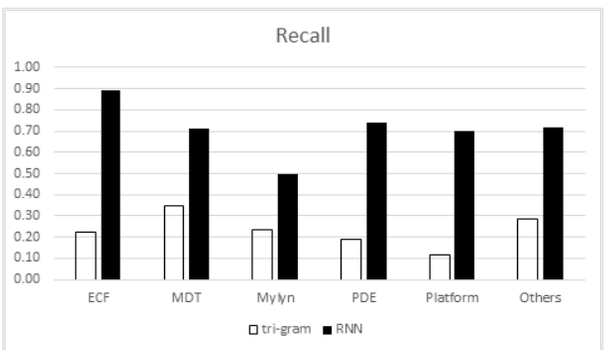

Fig. 7. Recall of N-gram and RNN

6. 실험 결과의 질적 조사

본 절에서는 우리의 모델의 추천 결과가 실질적으로 어떤 의미를 가질 수 있는지에 대한 조사를 진행한다. 이를 위해 2절 연구동기에서 사용한 두 개의 버그리포트에 대한 추천 결과가 가지는 의미에 대한 조사를 진행한다.

먼저 Platform 프로젝트의 100799번 버그리포트의 동작기록에서 보면 첫 번째 수정이 여섯 번째 동작에서 발생하는데 이는 처음 탐색 시간으로부터 약 9분 후 발생한다. 두 번째 수정은 여덟 번째 동작에서 발생하는데 이는 처음 탐색으로부터 약 10분 후 발생한다. 우리의 추천 방법을 사용하여 수정해야 하는 파일을 추천할 때, 처음 추천은 세 번째 탐색 동작이 발생할 때 바로 수정해야 하는 첫 번째 파일을 추천한다. 우리의 추천 방법을 적용하면, 첫 번째 수정파일을 첫 번째 탐색에서부터 세 번째 탐색이 시작되는 약 5분 후에 제시해 줄 수 있다. 그 결과, 바로 여섯 번째 동작으로 유도함으로써 기존의 네 번째, 다섯 번째 동작을 뛰어넘을 수 있다. 또한 위의 순서대로 진행하여 여섯 번째에 수정해야 하는 첫 번째 파일이 모델에 입력되면 다음 수정해야 하는 두 번째 파일을 추천한다. 이로써 첫 번째 수정에서 5분의 시간을, 두 번째 수정을 위해서는 1분의 시간을 단축할 수 있다. 이를 통해 예측한다면 실제 약 2시간 30분 걸렸던 탐색 및 수정 작업의 시간을 약 1시간 05분으로 약 56% 단축시킬 수 있다고 추정할 수 있다.

다음으로 동일 프로젝트의 193832번 버그리포트의 동작기록에서 첫 번째 수정이 세 번째 동작에서 발생하는데 이는 처음 탐색 시간으로부터 약 1분 후에 발생한다. 두 번째 수정이 스무 번째 동작에서 발생하는데 이는 처음 탐색 시간으로부터 약 11분 후에 발생한다. 우리의 모델은 첫 번째 추천을 발생시킬 때 반드시 세 번의 동작기록이 입력되어야 하기 때문에, 세 번째에서 발생한 첫 번째 수정파일은 추천할 수 없다. 하지만 첫 번째 수정을 진행하면 세 번의 동작기록이 입력되기 때문에 바로 다음 수정해야 하는 파일을 추천하여 열여덟 번의 탐색을 뛰어넘어 약 11분의 시간을 단축할 수 있다. 이를 통해 예측한다면 실제 약 23분 걸렸던 탐색 및 수정 작업의 시간을 약 13분으로 약 56% 단축시킬 수 있을 것으로 기대한다.

실질적 조사의 결과 우리의 모델은 전체 작업 시간을 약 56% 단축할 수 있을 것으로 기대된다. 특히 한 번의 수정작업 이후 다음 수정작업까지 걸리는 탐색의 횟수가 많을수록, 그 시간이 오래 걸릴수록 효과적이다. 그러나 실제로는 코드를 이해하기 위한 탐색에 걸리는 시간은 반드시 필요할 것이므로 조사에 의한 단축 시간보다는 적을 것으로 예상한다. 하지만 탐색을 통해 수정해야 하는 파일을 찾는 것과 수정해야 하는 파일을 알고 이해해야 하는 파일을 탐색하는 것에는 분명 차이가 있기 때문에, 수정해야 하는 파일을 추천해주는 우리의 연구는 충분한 의미가 있다고 볼 수 있다.

7. 관련 연구

본 절에서는 우리의 연구와 관련된 연구들을 기술한다. 관련 연구들은 통계적 언어모델을 사용한 코드 추천에 관한 연구와 소프트웨어 공학 문제에 딥러닝을 적용한 연구, 동작기록에 마이닝을 적용한 연구들이 있었다.

먼저 통계적 언어모델을 사용해 코드를 추천한 많은 연구들은 주로 N-gram 모델을 사용하였다. 예를 들면 Raychev 등은 API를 사용하는 프로그램에서 API를 포함한 코드 블럭에 누락된 코드가 있을 경우 코드의 완성을 확률적으로 예측해 코드를 추천하는 시스템을 연구하였다[3]. 또 Nguyen 등은 기존의 N-gram은 코드의 어휘 정보만을 기반으로 학습한다는 한계점에 의해 떨어질 수 있는 정확도 문제를 해결하기 위해 의미를 이해할 수 있는 새로운 N-gram 모델을 제시하였다[4]. 그리고 Nguyen 등은 N-gram기법을 그래프 모델 기반에 적용하여 코드의 출현 확률을 계산하여 코드에서 누락된 부분을 완성시키기 위한 코드를 추천하는 시스템의 연구를 진행하였다[5].

다음으로 소프트웨어 공학 문제에 딥러닝을 적용하여 우수한 성능을 보인 연구 사례들이 많았다. 그중 우리는 추천 문제에 딥러닝을 적용한 사례들을 대상으로 조사하였다. 예를 들면 Gu 등은 Java 언어 APIs의 사용 순서와 설명을 딥러닝 모델 중 Recurrent Neural Network 모델에 학습시켜, 자연어 문장을 입력하는 것으로 그에 맞는 APIs의 사용 순서를 추천하는 연구를 진행하였다[6]. Gu 등은 코드 저장소에서 불특정 다수의 Java 프로젝트와 C# 프로젝트로 부터 APIs 사용 순서와 그에 연관된 설명을 RNN모델에 학습시켜, Java 언어로 구현된 프로젝트를 C# 언어로 번역하는 연구를 진행하였다[7]. Lee 등은 버그에 대한 설명과 버그를 해결한 개발자의 데이터를 딥러닝 모델 중 Convolutional Neural Network 모델에 학습시켜, 새로운 버그

에 대처할 수 있는 개발자를 추천하는 연구를 진행하였다[8].

마지막으로 동작기록에 마이닝을 적용한 연구들이 있었다. 예를 들면 Lee 등은 프로그래머의 동작기록에 클러스터링 기법을 적용하여 개발 작업과 관련된 소스 코드를 군집화 하는 연구를 진행하였다[2]. 또 Dyer 등은 거대한 규모의 소프트웨어 저장소에 있는 수많은 소프트웨어 프로젝트에서 소스 코드에 대한 세부 수준의 정보 및 전체 이력 정보를 마이닝하기 위한 새로운 프로그래밍 언어를 제시하였다[12]. 또한 Lee 등은 개발자의 동작기록을 소프트웨어의 개정기록을 기반으로 하는 추천 시스템에 적용하는 연구를 진행하였다[13]. 그리고 Damevski 등은 개발자가 개발 환경의 사용 패턴을 시간의 순서로 마이닝하여, 반자동적으로 동작기록을 생성하는 연구를 진행하였다[14].

이처럼 기존 코드 추천 연구는 통계적 언어모델 중 N-gram이 중심이 되어 코드 추천 연구가 진행이 되어 왔다. 하지만 N-gram은 학습되지 않은 몇 개의 데이터가 입력되는 것만으로 재현율이 낮아지는 한계가 있다. 그런데 소프트웨어 공학 문제에 딥러닝을 적용한 연구들은 상당한 수준의 성능을 보였다. 이러한 결과는 우리가 딥러닝을 사용하도록 하는 동기가 되었으며, 동작기록이 개정기록보다 더 많은 정보를 가지고 있어 딥러닝에 더 적합할 것이라 기대한다.

8. 연구 결과의 타당성에 대한 위협

우리의 실험의 타당성에 대한 몇 가지 위협이 존재한다.

내부적으로 우리는 각 프로젝트의 동작기록에서 전처리 이후 생성된 데이터 쌍의 70%를 랜덤으로 추출하여 학습 시켰고 나머지 30%를 테스트 데이터로 사용하였다. 만약 또 다른 데이터 쌍 70%를 랜덤으로 추출하고 나머지 30%를 테스트한다면 모델의 성능은 달라질 수 있다. 그러나 랜덤 추출이기 때문에 달라지는 성능의 차이는 그리 크지 않을 것으로 예상한다.

외부적으로 우리는 단지 소수 프로젝트의 동작기록에 대해서만 평가하였으며, 동작기록을 단순히 시간 순서로 전처리하였지만, 만약 하나의 작업 단위로 전처리를 진행한다면 모델의 성능은 달라질 수 있다. 또한 탐색 및 수정된 파일의 이름을 딥러닝 모델 내부에서 임베딩 계층을 통해 벡터화 하였다. 이 벡터는 파일의 이름을 대표할 수는 있지만, 그 파일이 어떤 파일인지에 대한 의미는 대표할 수 없다. 즉 파일의 의미를 대표하는 다른 기법을 사용한다면 성능이 달라질 수 있다. 또한 단 두 개의 계층만을 가진 우리의 모델은 매우 단순하다. 계층을 더 늘리거나 신경망의 깊이를 늘린다면 모델의 성능은 달라질 수 있다. 그리고 우리는 새로운 데이터에 대한 실험을 진행하지 못하였기 때문에 새로운 데이터를 사용해 모델을 평가하게 된다면 모델의 성능은 달라질 수 있다.

9. 결 론

소프트웨어 프로젝트에서 시간은 핵심 자원 요소 중 하나이다. 개발자들은 하나의 코드를 수정하는데 들이는 시간보다 코드를 탐색하고 이해하는데 더 많은 시간을 소모한다. 우리는 개발자가 코드를 수정하기 위해 파일을 탐색하는데 걸리는 시간을 줄이기 위한 개발자의 동작기록과 딥러닝을 사용한 추천방법을 제안한다. 우리는 5개의 프로젝트에 대한 동작기록과 RNN모델을 사용해 사용자가 3개의 파일을 탐색하거나 수정하였을 때 다음으로 코드를 수정해야하는 파일을 추천해 주는 연구를 진행하였다. 그 결과 우리는 평균 약 91%의 정확도와 약 71%의 재현율을 달성하였다. 이러한 결과는 우리의 연구가 동작기록을 딥러닝 모델에 적용 가능하다는 점과 동작기록과 딥러닝 모델을 사용해 파일 수준에서 수정이 필요한 위치를 추천할 수 있음을 보인다.

향후 우리는 더 우수한 성능을 낼 수 있는 방법과 모델구현에 대한 연구를 진행할 것이다. 예를 들어 단 하나의 모델로 다양한 프로젝트에 적용가능 하도록 동작기록을 작업단위로 군집화 하고, 각 파일의 의미를 대표할 수 있는 기법에 대한 연구를 진행할 것이다. 또한 CNN과 RNN을 함께 사용하는 연구 등 더욱 정확한 모델을 위한 조합 모델의 연구를 진행할 것이다.

References

[1] J. S. Cho and J. C. Park, “A Study on The Project Schedule Management System Development for Small Scale IT Companies,” *The KORMS*, pp.1264-1272. 2008.

[2] S. A. Lee, S. W. Kang, S. H. Kim, and M. Staats, “The Impact of View Histories on Edit Recommendations,” *IEEE Transactions on Software Engineering*, Vol.41, Issue 3, pp.314-330, 2015.

[3] V. Raychev, M. Vechev, and E. Yahav, “Code completion with statistical language models,” in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation,* Edinburgh, pp.419-428, 2014.

[4] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, “A statistical semantic language model for source code,” *in Proceedings of the Joint Meeting on Foundations of Software Engineering*, Saint Petersburg, pp.532-542, 2013.

[5] A. T. Nguyen and T. N. Nguyen, “Graph-based statistical language model for code,” in *IEEE International Conference on Software Engineering*, Florence, 2015.

[6] X. Gu, H. Zhang, D. Zhang, and S. Kim, “Deep API learning,” in *Proceedings of the ACM SIGSOFT International Symposium on Foundation of Software Engineering*, Seattle, pp.631-642, 2016.

[7] X. Gu, H. Zhang, D. Zhang, and S. Kim, “DeepAM: Migrate APIs with multi-modal sequence to sequence learning,” *in Proceedings of International Joint Conference on Artificial Intelligence*, Melbourne, pp.3675-3681, 2017.

[8] S. R. Lee, M. J. Heo, C. G. Lee, M. Kim, and G. Jeong, “Applying deep learning based automatic bug triager to industrial projects,” in *Proceedings of the Joint Meeting on Foundations of Software Engineering,* Paderborn, pp.926-931.

[9] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural Computation*, Vol.9, No.8 pp.1735-1780.
[10] J. Y. Chung, C. Gulcehre, K. H. Cho, and Y. Bengio, “Empirical evaluation of gated recurrent neural networks on sequence modeling,” arXiv preprint arXiv:1412.3555, 2014.
[11] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” arXiv preprint arXiv:1412.6980, 2014.
[12] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, “Boa: Ultra-large-scale software repository and source-code mining,” *ACM Transactions on Software Engineering and Methodology*, Vol.25, Issue 1, Article No.7, 2015.
[13] S. A. Lee, S. W. Kim, S. H. Kim, and M. Staats, “The impact of view histories on edit recommendations,” *IEEE Transactions on Software Engineering*, Vol.41, No.3, pp.314-330, 2015.
[14] K. Damevski, D. C. Shepherd, J. Schneider, and L. Pollock, “Mining sequences of developer interactions in visual studio for usage smells,” *IEEE Transactions on Software Engineering*, Vol.43, No.4, pp.359-371, 2016.

조 희 태
https://orcid.org/0000-0002-1178-7852
e-mail : cht3205@gmail.com
2017년 경상대학교 항공우주 및 소프트웨어공학전공(학사)
2018년~현 재 경상대학교 정보과학과 석사과정
관심분야 : 소프트웨어 공학, 기계학습

이 선 아
https://orcid.org/0000-0002-2004-2924
e-mail : saleese@gnu.ac.kr
1997년 이화여자대학교 전산학(학사)
1999년 이화여자대학교 전산학(석사)
2005년 카네기멜론대학교 소프트웨어 공학(석사)
2013년 KAIST 전산학(박사)
1999년~2006년 삼성전자 선임/책임연구원
2013년~2015년 KAIST 연구 조교수
2016년~현 재 경상대학교 항공우주 및 소프트웨어공학전공/정보과학과 조교수, 항공기부품기술연구소 멤버
관심분야 : Software Architecture & Software Repository Mining & Recommendation System & Software Evolution

강 성 원
https://orcid.org/0000-0001-7947-8741
e-mail : sungwon.kang@kaist.ac.kr
1982년 서울대학교 사회과학대학(학사)
1989년 Univ. of Iowa 전산학(석사)
1992년 Univ. of Iowa 전산학(박사)
1993년~2001년 한국통신 연구개발본부 선임연구원
1995년~1996년 미국국립표준기술연구소(NIST) 객원연구원
2001년~2005년 한국정보통신대학교 조교수
2005년~2009년 한국정보통신대학교 부교수
2009년~현 재 KAIST 전산학부 부교수
관심분야 : 소프트웨어 아키텍처, 소프트웨어 제품라인, 소프트웨어 시험