공학 석사학위논문

추상 구문 트리의 단말 노드 간 경로에 타입 정보를 추가한 특징을 사용하는 딥러닝 기반 코드 이해

전남대학교 대학원
전자컴퓨터공학과

임 진 택

2020년 8월

추상 구문 트리의 단말 노드 간 경로에 타입 정보를 추가한 특징을 사용하는 딥러닝 기반 코드 이해

이 논문을 공학 석사학위 논문으로 제출함

전남대학교 대학원
전자컴퓨터공학과

임 진 택

지도교수 최 광 훈

임진택의 공학 석사의 학위논문을 인준함

심사위원장 전남대학교 부교수 공학박사 김 경 백 (인)

심사위원 전남대학교 조교수 공학박사 유 석 봉 (인)

심사위원 전남대학교 부교수 공학박사 최 광 훈 (인)

2020년 8월

목 차

그 림 목 차

표 목 차

추상 구문 트리의 단말 노드 간 경로에 타입 정보를 추가한 특징을 사용하는 딥러닝 기반 코드 이해

임 진 택

전남대학교 대학원 전자컴퓨터공학과

(지도교수 : 최 광 훈)

(국문초록)

최근 StackOverflow 및 Github와 같은 오픈소스 시장의 활성화로 인하여 온라인상에서 다양한 코드를 검색하고 프로그램을 작성하는 사례가 증가하고 있다. 하지만 현재는 제한된 패턴의 검색어로만 코드를 검색할 수 있다. 즉, 키워드로 제시한 패턴과 일치하지 않는 코드는 검색할 수 없다. 이와 같은 어려움은 오픈소스를 활용하는 소요시간을 증가시키는 결과를 얻어왔으며 이를 해결하기 위하여 코드의 특징을 추출하여 분석하는 연구가 필요하다.

한편, 인공지능 분야에서 데이터 기반으로 정교한 패턴을 찾는 딥러닝 기술이 각광받고 있다. 특히, 딥러닝 기술은 데이터의 양이 증가할수록 더욱 정교한 결과를 얻는데 이는 대규모의 소스코드를 보유한 오픈소스 시장에서 활용되기 유용한 장점을 갖고 있다. 소스코드를 딥러닝 모델에 학습함으로써 각 소스코드가 갖는 특징을 추출할 수 있으며 이를 통해 오픈소스의 환경을 개선할 수 있다.

본 논문에서는 딥러닝 기술을 활용하여 소스코드를 분석하는 개선된 코드 특징 추출 방법을 연구한다. 이와 관련된 연구들 중 우리는 추상 구문 트리(AST)의 경로를 이용하여 특징을 추출하는 code2seq 연구를 기반으로 삼아 연구를 진행한다. 추상 구문 트리는 코드의 구문 분석을 통해 구성되기 때문에 코드의 각 토큰간 관계를 이해할 수 있으며 이는 코드의 특징을 추출하는데 유리하다.

우리는 추상 구문 트리로 얻은 코드 정보에 타입 정보를 추가함으로써 데이터를 고도화하여 특징을 추출하는 성능을 향상시킨다. 소스 코드에서 타입 정보는 식별자의 특징을 나타내며 이는 곧 코드 전체 내용의 특징에 직결된다. 따라서 추상 구

문 트리의 경로 정보에 타입 정보를 추가한 데이터는 딥러닝 모델이 코드의 특징을 추출하는데 더 좋은 결과를 얻을 수 있게 한다.

이를 이용하여 두 가지 작업을 한다. 첫 번째 작업은 함수명 예측 작업이다. 함수명 예측 작업은 함수 코드를 통해 함수명을 예측하는 작업이며 이를 통해 함수명 규칙을 통일화할 수 있으며 개발 속도를 향상시킬 수 있다. code2seq에서 사용한 데이터 셋을 사용하여 기존 연구와 성능을 비교했을 때 타입 정보를 추가한 데이터 형식이 더 빠른 최적값 수렴과 더 높은 F1 점수를 얻을 수 있었다. 두 번째 작업은 함수 요약 작업이다. 함수 요약 작업은 함수의 코드를 통해 한 줄의 문장으로 요약하는 작업이며 이를 통해 오픈 소스를 사용하는 시간을 단축시킬 수 있다. 데이터 셋은 JavaDoc 스타일의 주석을 이용한 CONCODE 데이터 셋을 사용하였으며 함수 코드의 특징을 추출하여 한 줄의 문장으로 요약이 가능하다는 것을 보였다.

1. 서론

오늘날 전 세계적으로 많은 산업분야에서 IT 기술은 없어서는 안 될 기술 중 하나가 되었으며 기술 발전을 위해 프로그래머의 개발 환경 개선이 꾸준히 연구되고 있다. 개발자에게 있어서 개발 환경의 발전은 더욱 향상된 개발 속도와 융통성 있는 프로그래밍 패턴으로 직결될 수 있으며 오픈소스 시장의 활성화는 개발 환경을 개선하는 배경 중 하나가 되었다. 오픈소스 시장의 활성화는 코드 재사용을 통하여 개발비용을 절감할 수 있는 긍정적은 효과를 얻었으며 다양한 개발자들이 쉽게 접근할 수 있기 때문에 많은 개발자 층을 형성할 수 있게 되었다. 뿐만 아니라 오픈소스를 다양한 개발자들이 쉽게 공유할 수 있기 때문에 다양한 개발자들의 프로그래밍 패턴을 분석할 수 있다. 따라서 거대해진 오픈소스 데이터는 개발 환경을 개선하는데 사용될 수 있는 장점이 있다.

한편 최근 4차 산업혁명에 따라 인공지능 기술은 빠르게 성장하였으며 다양한 인간 분야에서 확고하게 자리를 잡고 있다. 특히 인공지능 기술은 다양한 분야에 융, 복합이 가능하다는 장점을 갖고 있으며 각 분야에서 축적된 데이터를 기반으로 패턴 및 특징을 추출할 수 있다. 이러한 인공지능 특징은 프로그래밍 언어 분야에 접목시키는데 큰 장점이 된다. 현재 가장 큰 오픈소스 리파지토리인 Github를 비롯하여 많은 오픈소스 리파지토리가 있으며 그동안 축적된 거대한 오픈소스 코드들은 인공지능분야에 데이터로 사용될 수 있다. 즉, [그림 1]과 같이 대규모의 소스코드는 인공지능 모델을 통해서 특징을 추출하여 코드를 분석할 수 있다.

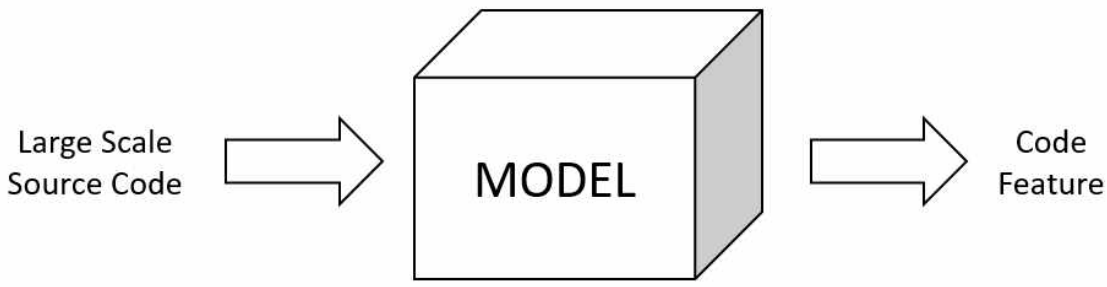


그림 1 대규모 소스코드를 이용한 딥러닝 기반 코드 분석

이러한 환경을 바탕으로 과거부터 인공지능 모델을 이용하여 코드를 분석하는 다양한 연구가 진행되어왔다. 특히 인공지능 기술 중 머신러닝(Machine Learning) 기술이 가장 주목을 받아 왔으며 이를 이용한 다양한 시도가 있었다. 예를 들어, 2001년 통계적 분석(Statistical analysis) 방법 기반의 머신러닝을 통해 템플릿-기능 쌍을 찾아 버그를 탐지하는 연구를 한 [1]부터 2013년 정보량을 확률적으로 분석하는 정보이론(Information Theory)을 통해 오픈소스 리파지토리의 패턴 분석 연구를 한 [2]까지 머신러닝을 이용한 다양한 코드 분석 연구가 진행되어왔다. 이에 나아가 최근 머신러닝의 일종인 딥러닝(Deep Learning)이 높은 성능으로 크게 주목받고 있으며 이에 따라 딥러닝을 이용한 코드 분석 연구 역시 최근 꾸준하게 진행되고 있다.

이러한 머신러닝과 딥러닝을 이용한 소스 코드 분석연구는 최근 함수명 예측과 같은 작업을 통해 성능을 증명하고 있으며 향후 오픈소스 시장의 검색기능을 포함한 다양한 기능을 제공할 수 있다. 즉, 함수명 예측 작업과 같이 어떠한 특정 함수의 소스 코드를 분석함으로써 소스 코드가 갖고 있는 코드의 특징을 키워드 중심으로 추출하고 이 키워드는 곧 코드의 특징들을 나타낸다. 이처럼 코드의 특징을 나타내는 키워드 집합을 통하여 향후 오픈소스 사용자는 좀 더 자유로운 키워드를 통하여 코드를 검색하는데 영향을 줄 수 있다. 따라서 소스코드 특징 추출 연구는 오픈소스 시장의 성장을 위해서 꼭 필요한 연구이다.

대규모 소스 코드의 특징을 추출하는 다양한 방법이 있지만 우리는 추상 구문 트리(Abstract Syntax Tree)의 구문간 관계를 잘 표현하는 장점을 이용하여 코드를 분석한다. 특히, 우리는 기존에 추상 구문 트리를 이용하여 소스코드를 분석한 다양한 연구 중 추상 구문 트리의 경로를 기반으로 한 code2seq[3]의 아이디어를 적극 수용하여 연구를 진행한다. 기존 연구는 추상 구문 트리의 단말 노드에서 다른 단말 노드까지 도달하는데 거치는 노드들로 구성된 모든 경로를 구하고 이를 딥러닝 모델에 입력 값으로 사용하여 소스코드의 특징을 추출한다. 우리는 여기에 더 나아가 각 경로에 타입 정보를 추가하여 데이터를 더 고도화한다. 소스코드에서 타입 정보는 각 식별자의 특징을 나타내며 소스코드는 이러한 다수의 식별자들로 구성되어 있다. 따라서 식별자의 특징을 잘 추출하면 코드 전체 내용의 특징을 추출하는데 직결된다. 우리는 이러한 접근으로 본 연구에 다가갔으며 기존 연구보다 더 향상된 결과를 보여준다.

우리는 두 가지 작업을 통해 기존 연구와 비교하여 성능을 입증한다. 첫 번째 작업은 함수명 예측이다. 함수의 소스코드 특징을 파악하여 일반적으로 개발자들이 사용하는 함수명을 예측하며 [3]에서 사용한 데이터 셋을 이용한다. 이 데이터 셋은 현재 가장 큰 오픈소스 리파지토리인 Github에서 Java 소스 코드를 수집한 데이터 셋으로 660MB의 Small 데이터 셋부터 15GB의 Large 데이터 셋이 있다. 우리는 이 데이터 셋을 이용하여 기존 데이터 형식의 결과와 우리가 제안하는 데이터 형식의 결과를 비교하여 타입 정보의 효과를 보인다.

두 번째 작업은 함수 요약이다. 함수명 예측 작업과 유사하게 함수의 소스코드 특징을 파악하여 한 문장으로 요약하며 [4]에서 제공하는 CONCODE 데이터 셋을 이용한다. 이 데이터 셋은 Github에서 Java 소스 중 JavaDoc 스타일의 태그가 포함된 코드들을 수집하여 각 코드마다 해당 태그를 매칭 시킨 데이터 셋으로 구성되어 있다. 여기서 JavaDoc은 Java 코드에서 API 문서를 HTML 형식으로 생성해주는 도구로써 JavaDoc 태그는 JavaDoc만의 주석 스타일이다. 따라서 Github에서 각 코드의 요약된 문장을 수집하기 유리하며 이를 이용하여 딥러닝 모델을 학습하고 새로운 코드가 들어왔을 때 학습된 모델을 기반으로 한 줄의 요약된 문장을 만들어낸다.

본 논문의 구성은 다음과 같다. 2장에서 최근 머신러닝(Machine Learning)을 이용하여 소스코드 분석을 진행한 연구들을 분석하고 본 연구의 모토가 된 code2seq에 대해서 분석한다. 3장에서는 우리가 제안하는 타입 정보를 추가하여 소스코드를 분석하는 방법에 대해서 설명한다. 4장에서 기존 모델과 비교를 통해 성능을 측정하고 논의사항을 말하며 5장에서 결론을 맺고 향후 연구에 대해서 설명한다.

2. 관련 연구

최근 인공지능 기술의 성장에 따라 프로그래밍 언어 학계에서 머신러닝을 이용한 코드 자동 완성, 결함 예측 등 다양한 연구가 진행되고 있다. 본 논문에서는 다양한 연구 중 소스코드 분석 및 특징 추출에 초점을 맞춘다. 소스코드 분석에 머신러닝이 도입되기 전까지 [5]와 같이 대규모 소스코드를 기반한 코드 특징 추출 개발 도구를 자체적으로 개발하여 코드 특징 추출 연구가 진행되어 왔지만 최근 머신러닝 기술의 성장에 따라 머신러닝 기반의 코드 특징 추출 연구에 집중되어 진행되고 있다. 따라서 본 장에서는 머신러닝을 이용한 소스코드 분석 중 최근에 진행된 연구를 설명하고 연구들 중 본 논문의 기반이 된 추상 구문 트리 경로를 이용한 특징 추출 모델과 모델에 쓰인 딥러닝 기술에 대하여 설명한다.

가. 머신러닝을 이용한 소스코드 분석

[2]는 Github에서 수집한 Java 소스코드를 사용하여 코드의 패턴을 찾아 프로젝트의 코드를 분석한다. 코드의 특징을 추출하기 위해 구체적인 코드 스니펫(code snippet)을 구해야하며 이를 위해 코드를 토큰단위로 나누어 이를 머신러닝 언어 모델인 n-gram과 정보이론을 이용하여 확률론적으로 접근하였다. 여기서 정보이론이란 정보의 양을 측정하는 응용 수학의 한 분야로 낮은 빈도의 상황에 더 많은 정보량의 가중치를 둠으로써 최대한 많은 정보를 얻어 정교한 확률 분포를 구하는 확률론적 접근 방식이다. 이는 최근 인공지능 기술 중 가장 많이 쓰이는 신경망에서 사용되는 손실함수의 기반이 되는 기술 중 하나이다.

[6]은 기존의 연구들과 다르게 딥러닝 모델을 사용하여 소스코드를 분석한다. n-gram과 같은 언어 모델을 사용할 시 종속 요소가 멀리 흩어져 있는 긴 코드의 특징을 잡지 못한다는 단점을 해결하기 위해 심층신경망 구조의 LSTM(Long Short Term Memory) 모델을 사용하여 코드의 전체적인 내용에 종속성을 더하여 분석한다.

[7]은 코드를 단지 토큰단위로 나누는 것이 아닌 추상 구문 트리를 만들어 학습 데이터로 이용함으로써 토큰 데이터들 사이의 관계 정보를 이해할 수 있는 모델을 제안하였다. 데이터의 형식이 트리 형식이기 때문에 이에 맞는 모델이 필요하였으며 이 논문에서는 대규모 데이터에 유리한 CNN(Convolutional Neural Networks)

을 트리 형식의 데이터를 학습할 수 있도록 고안하여 TBCNN(Tree-Based CNN)을 개발해 사용하였다.

[8]은 장거리 종속성 문제를 해결하기 위해 딥러닝 기반 그래프를 사용하여 코드의 구문과 의미 구조를 분석한다. 그래프는 GGNN(Gated Graph Neural Networks)을 사용하여 나타내며 그래프는 코드의 추상 구문 트리에 기초하여 구성된다. 그래프형식의 신경망을 사용함으로써 기존의 LSTM보다 종족성에 강하며 잘못된 변수명을 찾는 작업 또는 변수명을 지정하는 작업을 한다.

[9]와 [3]은 본 논문의 모토가 된 논문으로 소스코드를 분석해 함수명 예측 및 코드 요약과 같은 라벨링 작업을 한다. 핵심 아이디어는 추상 구문 트리의 가능한 모든 경로의 중요도를 구하여 Attention Mechanism을 사용해 학습에 이용하는 것이며 본 장 나.에서 좀 더 자세히 설명한다.

[10]과 [11]은 딥러닝 모델을 이용하여 JavaScript와 Python과 같은 동적 프로그래밍 언어 환경에서 변수 또는 함수의 타입을 유추하는 작업을 한다. 동적 프로그래밍 언어 특성상 각 변수와 함수의 타입은 동적으로 지정되므로 타입 안정성이 떨어지며 미흡한 IDE에 따라 여러 가지의 문제를 야기할 수 있다. 이를 위해 두 논문은 시퀀스 데이터에 강한 Bidirectional LSTM을 이용하여 타입을 유추하는 시도를 하였으며 [10]은 코드 전체의 내용을 통해 각 변수의 타입을 유추하고 [11]은 주석, 함수명 및 매개 변수명과 같은 자연어를 기반의 정보로 함수의 타입을 유추하는 작업을 수행한다.

이와 같이 소스코드 분석은 코드의 부분적 또는 전체적인 내용은 분석함으로써 식별자 이름 예측 및 타입 유추와 같은 작업을 수행할 수 있으며 이를 위해 다양한 방법으로 분석 능력을 향상시키기 위해 연구를 진행하는 것을 알 수 있다.

나. 추상 구문 트리 경로 기반 특징 추출 모델

앞 절에서 설명한 바와 같이 본 연구는 code2vec[9]와 code2seq[3]에 영감을 받아 진행하였으며 본 절에서는 두 연구 중 더 좋은 성능을 내는 code2seq에 대해서 설명한다. 1)에서 데이터의 기반이 되는 추상 구문 트리의 개념에 대해서 설명하고 2)와 3)에서 code2seq의 데이터 전처리 및 모델에 대해서 설명한다.

1) 추상 구문 트리(Abstract Syntax Tree)

우리가 흔히 사용하는 C언어, Java, Python 등과 같은 고급언어로 작성된 코드는 기계가 이해할 수 있는 기계언어로 번역이 되어야하며 번역되는 중간과정 중 하나가 추상구문트리(AST)이다. 추상 구문 트리는 코드의 구문을 분석한 파스 트리(Parse tree)의 일종으로 파스 트리에서 세미콜론과 같은 분리자 단말자 또는 하위 트리가 하나뿐인 사소한 루트 비단말자와 같이 불필요한 정보를 제외하고 필수적인 정보만 담은 트리이다. 불필요한 정보를 제거하였기 때문에 구문구조가 간결하게 표현되며 구문분석과정에서 Syntax-directed 방법으로 생성해내기 때문에 쉽게 트리를 구성할 수 있다.

추상 구문 트리 생성 과정은 [그림 2]와 같다. 소스코드가 들어오면 어휘 분석기(Lexer)를 통해서 의미가 있는 토큰 단위로 나누어 토큰 리스트로 변환한다. 이 변환 과정에서 주석을 포함한 모든 공백문자는 버려진다. 생성된 토큰 리스트는 구문 분석기(Parser)를 통해서 추상 구문 트리를 만든다. 구문 분석기는 연속으로 나열된 토큰을 읽어서 문법 규칙에 따라 추상 구문 트리를 생성한다.[12]

그림 2 추상 구문 트리 생성 과정

[그림 3]은 대입식 “z = x + 2 * y”를 추상 구문 트리로 변환의 예제를 보여준다. 대입식은 어휘 분석기를 통해 [“z”, “=”, “x”, “+”, “2”, “*”, “y”]의 토큰 리스트로 변환이 되며 이 토큰 리스트는 구문 분석기에서 토큰의 의미에 따라 추상 구문 트리를 생성한다. 추상 구문 트리의 각 노드들이 의미 규칙(Semantic Rule)에 따라 구성되기 때문에 목적코드 생성을 위한 의미 해석에 유리하다. 즉, 추상 구문 트리는 코드의 구문을 이해하고 특징을 추출하는데 유리하며 이 장점은 대규모 소스코드를 기반으로 한 소스코드 분석 연구 분야에서 강력한 힘을 보여준다.

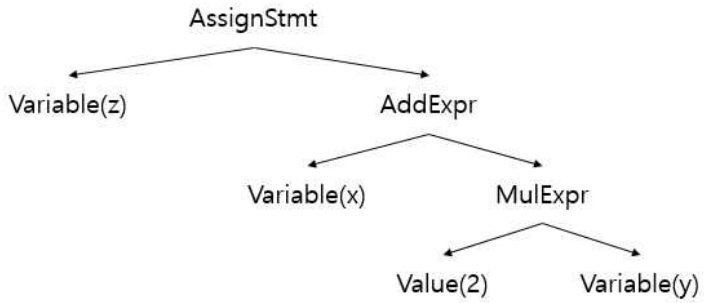


그림 3 대입식 “z = x + 2 * y” 의 추상 구문 트리

2) code2seq 경로 추출

딥러닝 모델을 사용하기 위해서 입력 데이터와 타겟 데이터가 필요하다. 추상 구문 트리를 이용한 코드 특징 추출 모델은 각 함수에서 나오는 추상 구문 트리를 입력 데이터로 사용하며 그에 따른 타겟 데이터로 함수의 라벨을 사용한다. code2seq의 경우 추상 구문 트리 데이터를 경로의 집합 형식으로 전처리 후 입력 데이터로 사용한다. 여기서 경로란 추상 구문 트리에 구성된 임의의 단말노드(Terminal Node)에서 시작하여 다른 단말노드에 도달할 때까지의 모든 노드들의 집합을 뜻하며 경로의 개념은 [13]에서 처음 제안이 되었다. 추상 구문 트리의 경로를 사용하여 특징을 추출하는 모델의 장점은 아래와 같다.

- 각 코드의 경로는 자동으로 생성되기 때문에 사용자는 각 프로그램 사이의 잠재적인 관계를 수동적으로 포착하여 사용할 필요 없이 간편히 사용할 수 있다.
- 추상 구문 트리를 이용하여 파싱하는 어느 언어에서도 사용하는데 큰 어려움이 없다.
- 추상 구문 트리의 경로는 단순히 구문(노드)들의 집합이며 각 노드의 의미 분석할 필요가 없다.

따라서 추상 구문 트리의 경로를 이용한 특징 추출 모델은 다른 언어로의 이식성이 높은 장점이 있으며 각 구문의 의미를 파악할 필요 없기 때문에 학습 데이터로 사용하기에 효율적이다. 예를 들어 ‘ReturnStmt’ 노드의 경우 리턴을 수행하는 명령문을 의미하지만 이러한 의미는 딥러닝 모델이 스스로 학습하기 때문에 사용자가 직접 입력할 필요가 없어 효율적인 데이터를 사용할 수 있다. 또한 나아가 구문들

의 집합인 경로를 사용하기 때문에 구문들 사이의 관계를 데이터로 사용함으로써 모델의 정확도를 향상시키는데 영향을 미칠 수 있다.

[그림 4]는 code2seq에서 Java 코드의 추상 구문 트리 경로를 추출하는 예제를 보여준다. (b)는 (a)의 코드를 파싱하여 나온 추상 구문 트리이며 같은 색으로 연결된 노드들은 경로들 중 일부를 나타낸다. 이해를 위해 경로 추출하는 과정을 3단계로 나누어 설명한다.

Step 1

앞서 설명했듯이 경로는 단말노드에서 시작하여 다른 단말노드에 도달할 때까지의 모든 노드들의 집합이다. 즉, 파란색으로 표현된 경로의 경우 아래와 같은 경로를 얻을 수 있다.

PrimitiveType -> MethodDeclaration -> BlockStmt -> ReturnStmt -> NameExpr

Step 2

각 경로의 시작 노드와 끝 노드의 Value를 토큰으로 입력 데이터에 추가함으로써 경로의 의미를 더 부과한다. 즉, Step 1에서 구한 경로에 시작 노드 Value와 끝 노드 Value를 추가하여 $(Token_s, Path, Token_e)$의 형태인 컨텍스트로 만들어 입력 데이터에 더 의미를 부과하며 파란 경로의 컨텍스트는 아래와 같이 표현된다.

(int, PrimitiveType -> MethodDeclaration ->
BlockStmt -> ReturnStmt -> NameExpr, num)

Step 3

마지막으로 효율적인 학습을 위해 데이터의 크기를 최소화해야 할 필요가 있으며 추상 구문 트리의 노드는 한정적이라는 특성에 따라 각 노드는 축약된 이름을 사용할 수 있다. Javaparser의 경우 약 100개의 노드가 있으며 [표 1]은 code2seq에서 사용한 축약된 노드 이름의 일부이다. 따라서 학습에 사용되는 최종적인 파란 경로의 컨텍스트는 아래와 같다.

(int, Prim -> Mth -> Bk -> Ret -> Nm, num)

하나의 함수는 이와 같은 컨텍스트를 최소 하나 이상 만들어 낸다. [그림 4]의 경우 함수 countOccurrences는 총 15개의 단말 노드를 갖고 있기 때문에 단말 노드 튜플 개수는 ${}_{15}C_2$이며 총 105개의 컨텍스트를 만들 수 있다.

정리하면 n개의 단말 노드를 갖고 있는 하나의 함수는 함수 라벨링을 위해 총 ${}_nC_2$개의 컨텍스트를 딥러닝 모델의 입력 데이터로 사용하여 학습을 진행한다.

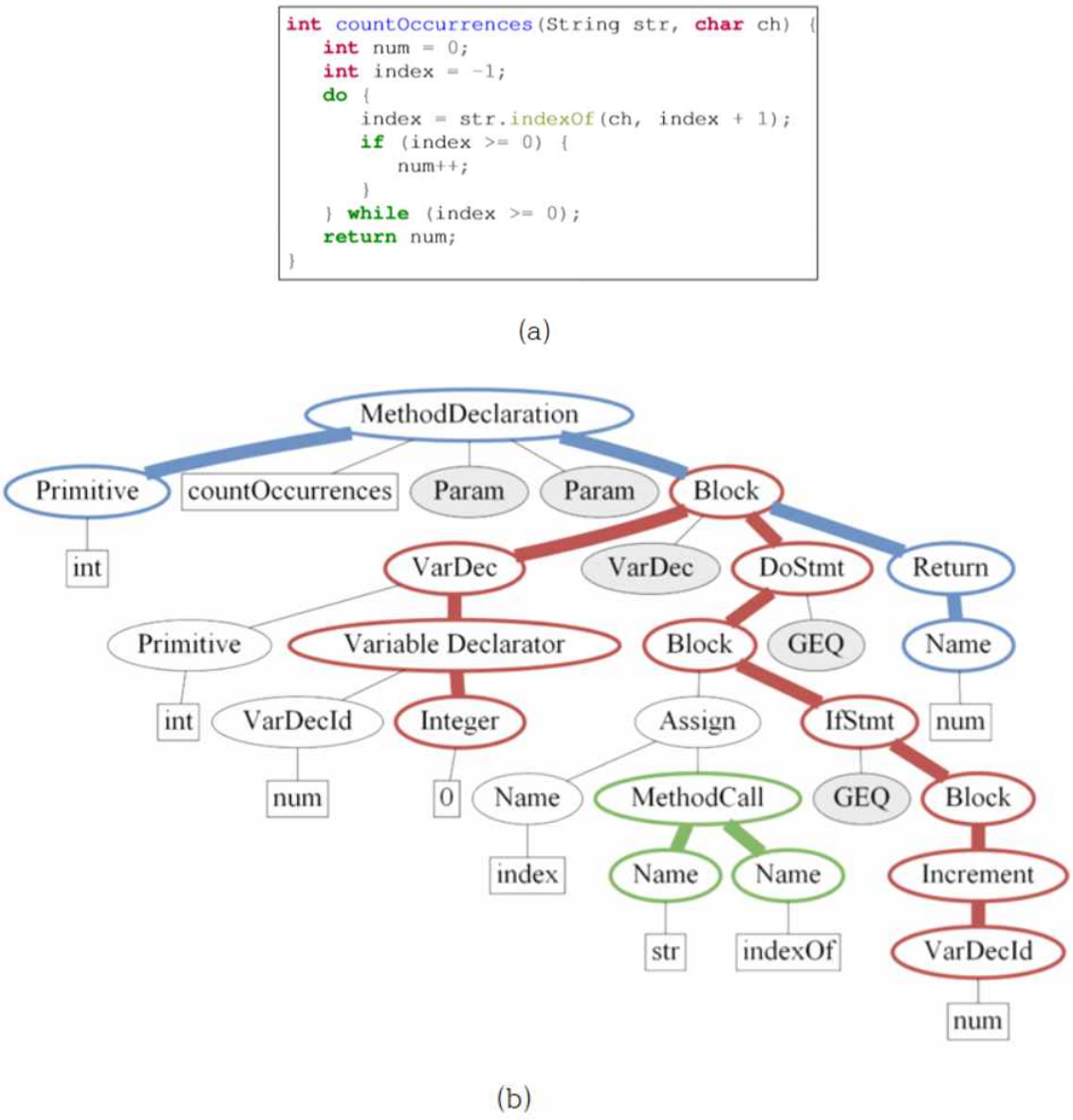


그림 4 code2seq에서 소스코드에 따른 추상 구문 트리와 생성 가능한 경로의 예

표 1 노드 이름을 축약시킨 이름들의 일부

노드 이름	축약된 노드 이름
AssignExpr:assign	As
AssignExpr:plus	AsP
BinaryExpr:binAnd	BinAnd
BlockStmt	Bk
ClassExpr	ClsEx
ClassOrInterfaceType	Cls
DoStmt	Do
IfStmt	If
LambdaExpr	Lambda
MethodCallExpr	Cal
MethodDeclaration	Mth
NameExpr	Nm
Parameter	Prm
PrimitiveType	Prim
ReturnStmt	Ret
StringLiteralExpr	StrEx
ThisExpr	This
TypeExpr	Type
VariableDeclarationExpr	VDE
VariableDeclarator	VD
WhileStmt	While
. . .	. . .

3) code2seq 모델

code2seq에서 사용한 모델은 [그림 5]와 같다. 앞의 2.2에서 설명했듯이 모델의 입력 데이터는 하나의 함수 코드마다 $(Token_s, Path, Token_e)$의 형태의 컨텍스트가 여러 개 들어온다. 즉, 그림의 입력층과 같이 하나의 컨텍스트는 시작 토큰, 경로, 끝 토큰의 형태로 입력을 받아들이며 이를 하나의 벡터로 만들어 학습 진행이 된다.

이처럼 하나의 컨텍스트는 하나의 벡터가 되고 이러한 컨텍스트 벡터들을 이용하여 Attention Mechanism과 Decoder를 통해 결과 시퀀스를 만들어 낸다.

본 연구는 code2seq의 모델을 사용하였기 때문에 전체적인 모델 구조 및 원리는 본 논문 3장 나. 에서 좀 더 자세하게 설명한다.

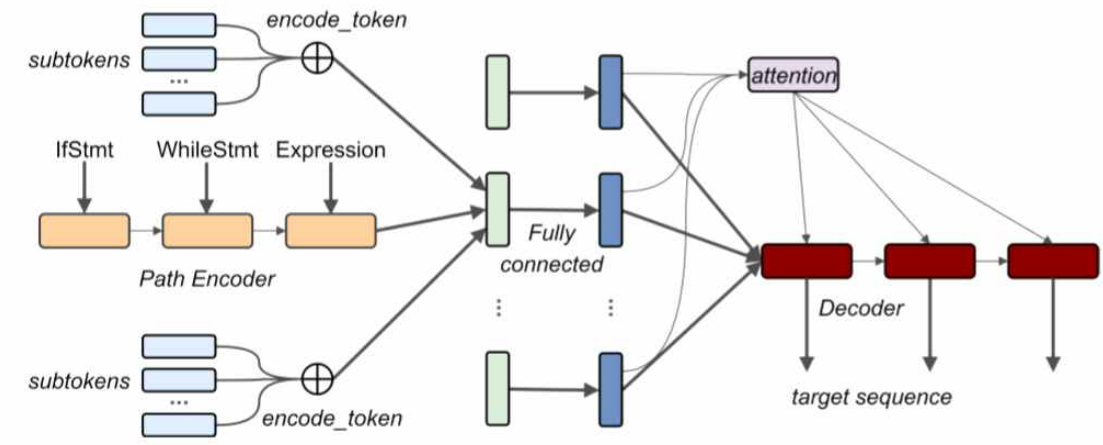


그림 5 code2seq의 모델 구성

다. 시퀀스 데이터를 위한 딥러닝 기술

이 절에서는 시퀀스 데이터를 다루기 위한 딥러닝 기술에 대해서 설명한다. 시퀀스 데이터는 순차적이고 길이가 가변적인 특징에 따라 이에 적합한 딥러닝 모델이 필요하다. 예를 들어, 텍스트에는 문맥이라는 순서가 있고, 시계열 데이터는 시간이라는 순서가 있다. 따라서 이러한 순서에 적합한 모델이 필요하며 1)에서 순차적 데이터에 강력한 LSTM을 설명하고 2)에서 시퀀스 데이터에서 시퀀스 데이터를 만드는 Seq2seq 모델의 원리인 Encoder-Decoder 구조를 설명하고 3)에서 최근 시퀀스 모델에 긍정적인 영향을 주었던 Attention Mechanism을 설명한다.

1) LSTM(Long Short Term Memory)

LSTM[14]는 1997년에 기존 RNN(Recurrent Neural Network)이 갖고 있던 장기 의존성 문제를 해결하기 위해 제안되었다. 기존 RNN인 Vanilla RNN은 시퀀스 데이터를 처리하는데 적합한 형태로 디자인 되었으며 순차 데이터를 쉽게 모델링할 수 있는 장점이 있다. 하지만 Vanilla RNN은 학습이 계속 진행됨에 따라 그레디언트 소멸 문제(Gradient Vanishing)가 발생하여 학습 데이터가 길어질수록 먼 과거의 정보를 현재에 전달하기 힘든 현상이 나타났다.

이를 해결하기 위해 LSTM이 제안되었다. LSTM의 구조는 [그림 6]과 같다. LSTM은 크게 기억소자(Memory Cell), 유지 게이트(Forget Gate), 입력 게이트(Input Gate), 출력 게이트(Output Gate)로 이루어져있다. LSTM의 핵심 아이디어는 기억소자를 통해 순차 데이터를 효율적으로 기억하고 관리하는 것이며 각 게이트를 통해 정보의 흐름을 조절한다. RNN의 원리와 같이 지난 은닉층의 출력을 받아 학습에 이용되며 이에 더해 기억소자를 이용하여 오래된 정보를 유지시켜 장기 의존성 문제를 해결하는 것이다. 각 게이트에는 가중치 W와 편향 b가 있으며 이 변수들은 딥러닝 훈련을 통해 학습이 된다. 즉, 훈련을 통해 각 게이트에서 어느 정도의 데이터를 유지시키고 어떠한 특징을 추출할지를 학습한다.

유지 게이트는 이전 정보에 대해 유지시키는 정도를 조절한다. 0과 1사이의 f_t값을 구해 이전 정보인 c_{t-1}과 곱하여 이전 정보의 일부를 버리게 된다. t-1시간의 은닉층에서 온 h_{t-1}와 현재시간 t의 입력 x_t에 sigmoid 활성화 함수를 적용하여 유지시킬 정도의 f_t를 구할 수 있으며 f_t는 아래와 같다.

$$f_t = \sigma(W_f \bullet [h_{t-1}, x_t] + b_f)$$

입력 게이트는 현재 입력 값에서 어떤 정보를 메모리에 기록할지 결정한다. 그림을 보면 입력 게이트는 입력 데이터에 sigmoid 활성화 함수를 적용한 i_t와 tnah 활성화 함수를 적용한 $\tilde{c_t}$를 곱하여 기억소자에 정보를 저장시킨다. 즉, 0과 1사이의 i_t를 통해 어떤 정보를 입력할지 정하고 -1과 1사이의 $\tilde{c_t}$를 통해 그 정보를 얼마나 저장할지 정하여 기억 소자에 저장하는 것이다. 입력 게이트에서 사용하는 식은 아래와 같다.

$$i_t = \sigma(W_i \bullet [t_{t-1}, x_t] + b_i)$$

$$\tilde{c_t} = \tanh(W_c \bullet [h_{t-1}, x_t] + b_c)$$

$$c_t = f_t * c_{t-1} + i_t * \tilde{c_t}$$

마지막으로 출력 게이트는 t시간에서 어떤 값을 출력할지 결정한다. 입력 데이터와 기억소자에 저장된 데이터를 기반으로 t시간의 출력 데이터를 만들어 내야한다. 따라서 t시간까지 기억소자에 기억된 데이터 c_t에 tanh 활성화 함수를 이용한 값과 입력 데이터에 sigmoid 활성화 함수를 적용한 값을 곱하여 출력 데이터를 만들어 내며 이 값은 다음 은닉층에 전달된다. 출력 게이트에서 사용되는 o_t와 h_t 및 y_t는 아래와 같다.

$$o_t = \sigma(W_o \bullet [h_{t-1}, x_t] + b_o)$$

$$h_t = y_t = o_t * \tanh(c_t)$$

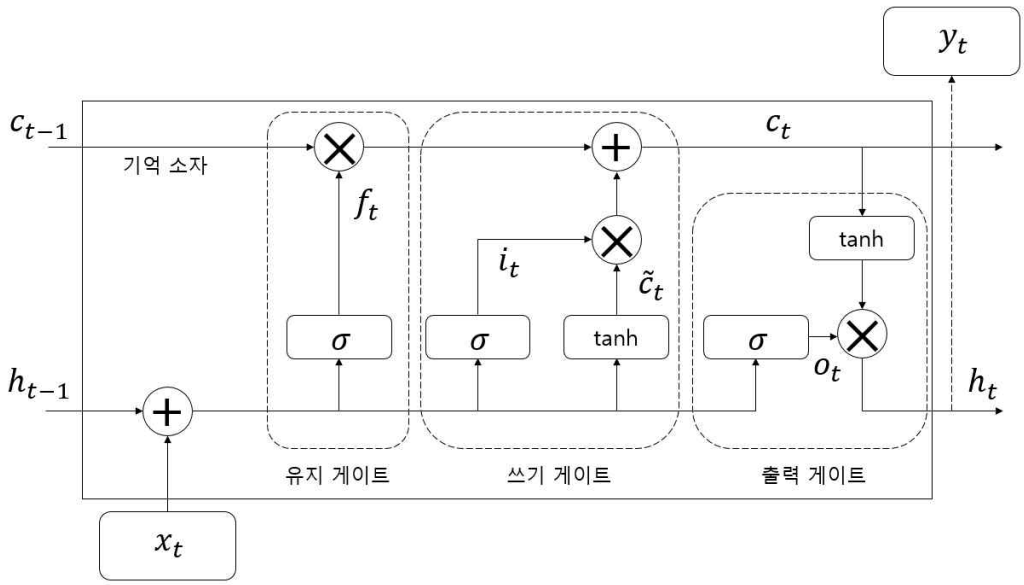


그림 6 LSTM(Long Short Term Memory) 구조

2) Encoder-Decoder

딥러닝 모델에서 길이가 가변적인 시퀀스를 입력으로 받아 다른 길이의 시퀀스로 출력하기 위해 seq2seq 모형을 사용한다. 예를 들어, [그림 7]은 seq2seq 모형의 가장 대표적인 예인 언어 번역(Language Translation)을 보여준다. 그림과 같이 한-영 번역기의 경우 “나는 내일 열리는 학회에 참석한다.”를 입력 데이터로 넣었을 때 “I will attend the conference held tomorrow.”의 출력을 내야한다. 예제를 보면 입력 데이터의 길이와 출력 데이터의 길이가 다른 것을 확인 할 수 있다. 이처럼 가변적인 특성을 지닌 시퀀스를 학습하기 위해 인코더와 디코더가 필요하다.

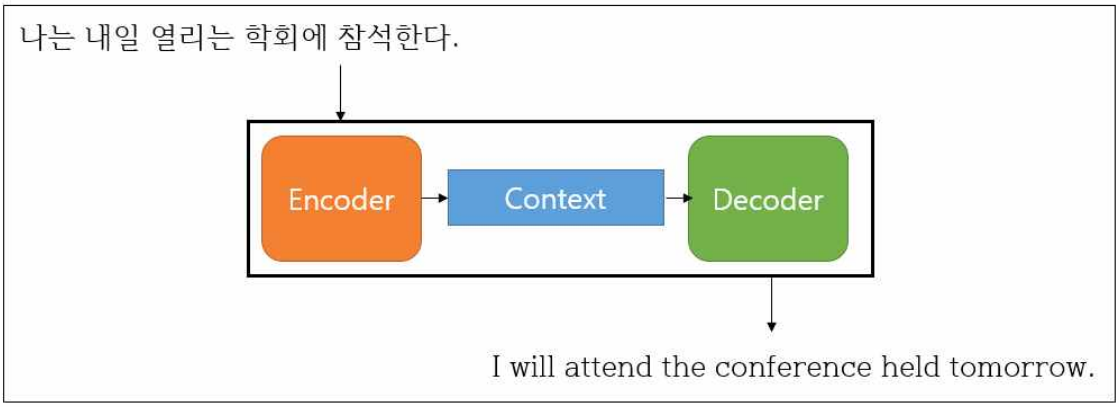


그림 7 seq2seq의 대표적인 모형인 언어 번역 모델

인코더는 입력 시퀀스를 특징이 잘 표현되는 하나의 벡터로 만드는 역할을 한다. 즉, 문장의 모든 단어들을 순차적으로 입력받은 후 순차적인 성질을 살려 단어 정보들을 압축해서 하나의 벡터로 만들어야 하며 문맥의 정보 포함하고 있다하여 이를 컨텍스트 벡터(Context Vector)라고 한다. 인코더를 통해 시퀀스를 정보가 압축된 컨텍스트 벡터로 만들었다면 디코더를 통해 번역된 단어를 한 개씩 순차적으로 출력해야한다.

입력 데이터와 출력 데이터가 순차적인 시퀀스 데이터이므로 3.1에서 설명한 바와 같이 LSTM을 사용하는 것이 가장 유리하다. 따라서 일반적인 seq2seq의 모델은 [그림 8]과 같이 LSTM기반 Encoder-Decoder를 사용한다. 순서가 있는 입력 데이터를 LSTM기반 인코더를 이용하여 하나의 컨텍스트 벡터로 만들고 디코더에서 단어 하나씩 출력해낸다. 디코더의 처음 <sos>는 시작을 의미하는 심볼이며 그 이후에는 다음에 등장할 확률이 가장 높은 단어를 예측한다. 즉, 그림 예제에서 디

코더의 첫 번째 LSTM 셀은 처음에 등장할 단어가 “I”일 것이라고 예측을 하였고 이 단어는 다음 두 번째 LSTM 셀의 입력으로 들어간다. 그리고 두 번째 LSTIM 셀은 입력된 단어 “I”와 첫 번째 LSTM에서 넘어온 h_1, c_1를 기반으로 다음에 올 단어를 “will”을 예측하는 것이다.

이처럼 시퀀스를 하나의 의미 있는 벡터로 압축하고 이를 통해 하나의 출력 시퀀스를 생성하는 구조를 Encoder-Decoder 구조라고 하며 벡터로 압축하는 구조를 인코더, 의미 있는 벡터를 하나의 시퀀스로 만들어내는 구조를 디코더라고 한다.

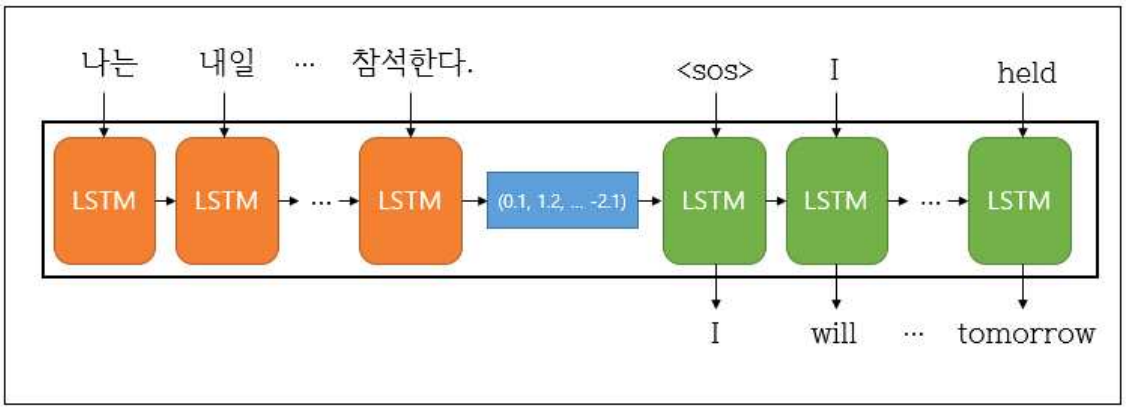


그림 8 LSTM 기반 seq2seq 언어 번역 모델

3) Attention Mechanism

3.2에서 설명했듯이 Encoder-Decoder 모델에서 중요한 특성은 인코더를 통해 하나의 시퀀스가 하나의 벡터로 압축된다는 것이었다. 하지만 이러한 구조는 인코더에서 계산한 여러 은닉층(그림 5에서 각 LSTM층) 중 마지막 은닉층만을 디코더에서 사용하게 되는 문제가 있었다. 즉, 컨텍스트 벡터는 문장의 앞부분의 정보는 희미하고 뒷부분의 정보가 강력하게 압축되는 단점이 존재하였다.

이를 해결하기 위해 2015년에 어텐션 메커니즘(Attention Mechanism)[15]가 제안되었다. 어텐션 메커니즘의 핵심 아이디어는 두 가지이다. 첫 번째 아이디어는 디코더에서 출력 결과를 예측하는 매 시점마다 인코더의 은닉층을 다시 한 번 참고한다는 것이다. 그리고 두 번째 아이디어는 예측하는 매 시점마다 예측해야하는 결과가 어떤 입력 은닉층에 가장 연관이 있는지 판단하여 비율을 매기고 그 입력 은닉층을 좀 더 집중하여 보며 이를 어텐션(Attention)이라고 한다.

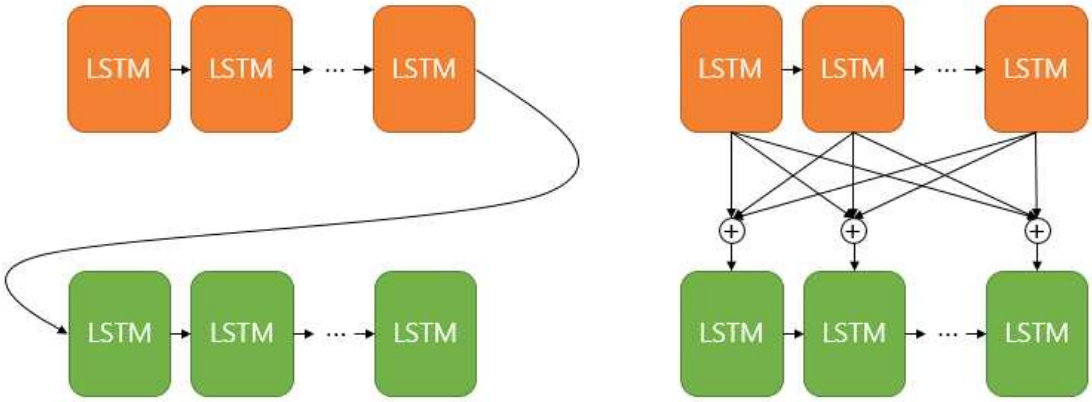


그림 9 기존 Encoder-Decoder와 어텐션 메커니즘

[그림 9]는 기존 Encoder-Decoder 구조와 어텐션 메커니즘의 차이를 보여준다. (a)와 같이 기존 Encoder-Decoder 구조는 디코더가 예측하기 전에 인코더의 최종 상태만을 이용하지만 어텐션 메커니즘을 이용하면 (b)와 같이 매 출력 시점마다 모든 은닉층을 참고하여 결과를 도출한다. 이를 통해 시퀀스의 앞부분 의미가 소멸하는 문제를 해결할 수 있다.

그리고 예측하는 매 시점마다 입력 은닉층의 비율에 따라 집중함으로써 결과의 정확도를 높일 수 있다. [그림 8]을 보면 각 단어의 위치가 매칭이 되지 않는다는 것을 확인할 수 있다. 즉, “내일”이라는 단어는 “tomorrow”에 매칭이 되지만 그 단어가 문장에 위치한 곳은 서로 다르다. 이처럼 결과의 위치와 별개로 매 시점의 결과는 각각 중요도를 보는 위치가 다르다. 이를 해결하기 위해 각 결과는 입력 단어의 위치와는 관계없이 각각의 중요도를 갖고 결과를 예측해야한다.

[그림 10]은 어텐션 메커니즘 기법 중 가장 기본적으로 사용되는 Dot-Product Attention 기법을 도식화한 것이다. 만약 입력 시퀀스로 3개의 단어가 순차적으로 들어왔을 때 그림과 같이 각 시점의 결과는 입력 데이터의 중요도를 측정하여 그에 따라 결과를 예측한다. 그림 예제의 경우 결과 시퀀스의 첫 번째 단어는 입력 시퀀스의 첫 번째 단어에 대해 가장 높은 중요도를 두고 예측을 했으며 결과 시컨스의 두 번째 단어는 입력 시퀀스의 세 번째 단어에 가장 중요도를 두고 예측을 한 것을 볼 수 있다. 이처럼 각 은닉층의 어텐션 분포를 구했다면 이를 각 은닉층의 입력 벡터와 어텐션 확률을 가중합하여 중요도에 비례한 결과를 낼 수 있다.

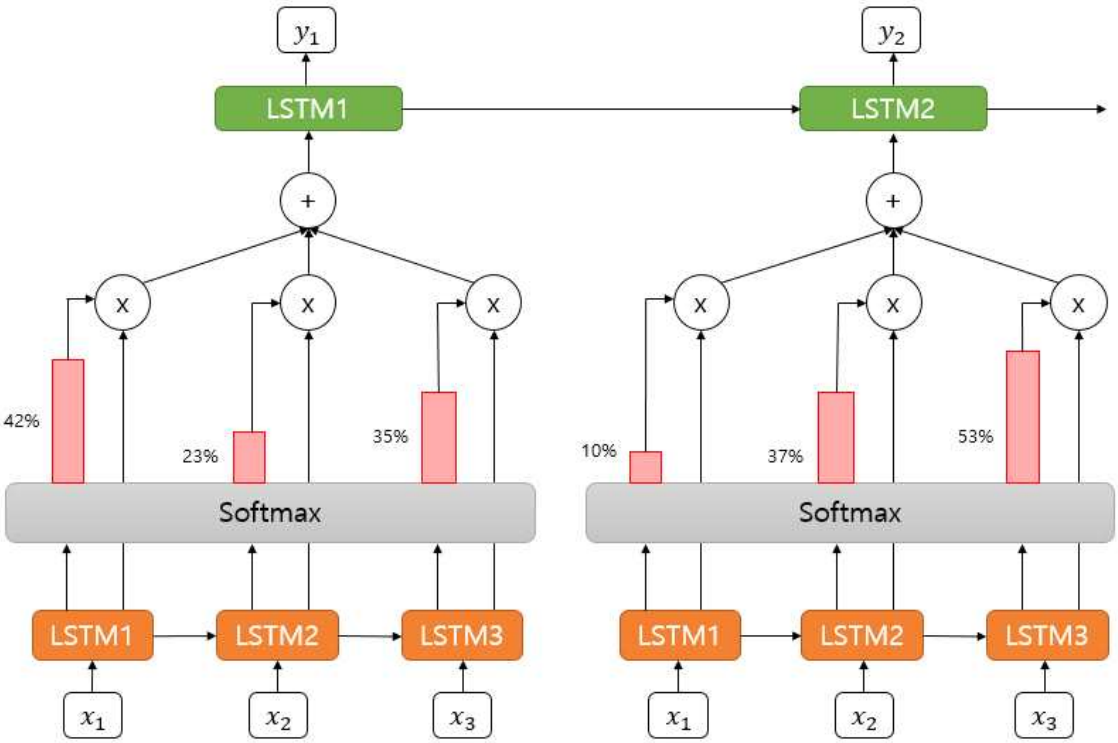


그림 10 어텐션 메커니즘 기법 중 Dot-Product Attention 기법의 구조

3. 타입 기반 소스코드 특징 추출 모델

프로그래밍 언어에서 타입 정보는 식별자의 특징을 잘 나타내며 식별자의 특징 추출은 곧 소스코드의 특징 추출에 직결된다. 우리는 이러한 특징을 이용하여 학습 데이터를 고도화함으로써 모델의 성능을 향상시킨다. 이를 위해 기존의 code2seq 연구의 아이디어를 수용하며 code2seq의 데이터 형식에 타입 정보를 추가하여 학습에 이용한다. 앞서 2장의 나에서 설명했듯이 code2seq의 경로를 이용한 데이터를 사용하면 각 구문의 의미 해석이 필요 없으며 딥러닝 모델은 각 구문의 관계를 통하여 각 구문의 의미를 이해한다. 따라서 경로의 각 노드간의 관계 정보의 고도화는 모델의 성능을 향상 시킬 수 있다. 우리는 경로에 타입 정보 노드를 추가함으로써 식별자 노드와 타입 정보 노드의 관계 정보를 통해 딥러닝 모델이 각 식별자 노드의 의미를 이해하는데 도움을 주며 모델의 성능을 향상시킨다.

타입 기반 소스코드 특징 추출 과정은 [그림 11]과 같다. 특징 추출 과정 중 첫 단계는 학습에 이용할 소스코드 데이터 셋을 준비하여야 한다. 우리는 모델을 적용할 수 있는 작업에 따라 두 가지 유형의 자바 소스코드 데이터 셋을 준비했으며 이에 관한 내용은 3절에서 설명한다. 수만 개에서 수십만 개의 데이터 셋이 준비가 되면 이 소스코드를 경로 추출기를 이용하여 데이터 전처리 과정을 해야 하며 이 과정은 [그림 11]에서 2번과 3번에 해당된다. 기존의 code2seq에서 사용한 경로 추출기와는 다르게 경로에 타입 정보를 함께 추출하여 전처리를 하며 이에 관한 내용은 가. 에서 설명한다. 소스코드를 컨텍스트 형태의 데이터로 전처리과정이 끝났다면 이를 이용하여 딥러닝 모델을 학습시킨다. 우리는 code2seq의 모델을 사용하였으며 모델의 작동 원리 및 과정에 대해서 나.에서 설명한다.

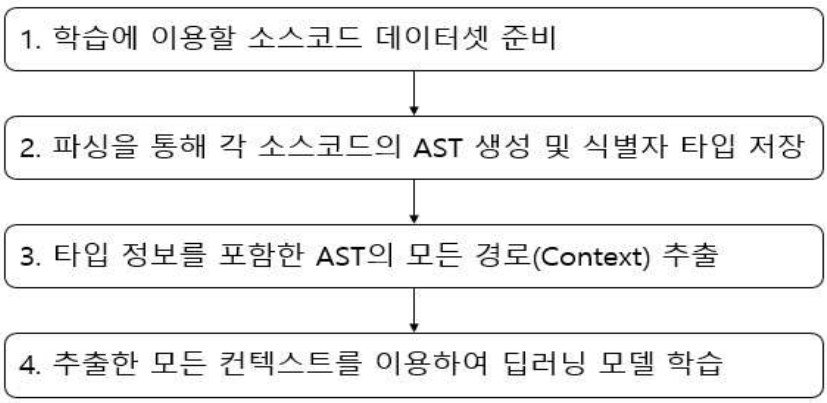


그림 11 타입 기반 소스코드 특징 추출 과정

가. 타입 정보를 포함한 데이터 전처리

1) 추상 구문 트리의 타입 정보

추상 구문 트리의 노드 중 일부는 각 노드가 지니고 있는 값에 따라 타입 정보를 갖고 있으며 이는 소스코드의 특징을 추출하는데 영향을 줄 수 있다. 본 절에서 자바 환경에서 추상 구문 트리의 타입을 추출하는 과정을 설명한다. [그림 12]는 자바에서 흔히 사용하는 파서인 Javaparser[16]을 이용하여 추상 구문 트리에서 타입 정보를 얻을 수 있는 예제를 보여준다. (a)는 학생들의 영어 성적을 관리하는 간단한 샘플 소스코드이며 (b)는 이 코드를 Javaparser를 이용하여 구한 추상 구문 트리와 타입 정보를 추출한 샘플 추상 구문 트리이다.

기존 code2seq는 함수 단위로 특징을 추출하기 때문에 MethodDeclaration 노드를 기준으로 그 아래의 서브 트리만 데이터로 사용하여 특징 추출을 한다. 따라서 함수 setLevel 안에 있는 Speaking 변수의 타입 정보를 얻을 수 없을 뿐만 아니라 단말 노드의 값만을 사용하기 때문에 경로 중간의 노드들의 타입 정보 역시 데이터로 사용할 수 없다.

우리는 기존의 경로 데이터에 타입 정보를 더하기 위해 (b)에서 빨간 글씨로 표현된 정보와 같이 각 노드가 갖고 있는 타입 정보를 추출하여 경로에 타입 정보를 추가하여 사용한다. 예를 들어, 파란색 경로의 경우 기존 code2seq에서는 2장 2절에서 설명한 방식에 따라 아래와 같이 컨텍스트를 구하여 데이터로 사용하였다.

(Speaking, Nm -> Cal -> Ex -> Bk -> Mth -> Prm -> VDID, Level)

우리는 이 컨텍스트를 기반으로 각 노드에서 추출할 수 있는 타입 정보를 추출하고 이를 경로에 추가하여 경로가 담고 있는 정보를 더 고도화한다. 따라서 우리가 제안하는 데이터 형식에 따르면 파란색 경로는 아래와 같이 변형하여 사용된다.

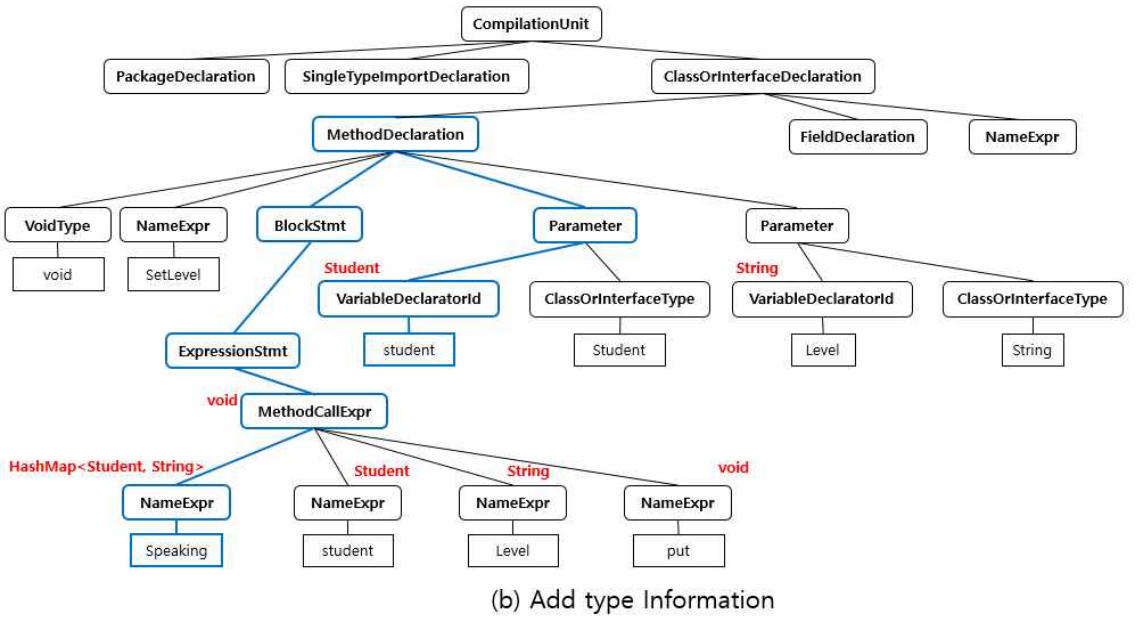
(Speaking, Nm | **HashMap<Student, String>** -> Cal | **void** ->
Ex -> Bk -> Mth -> Prm -> VDID | **String**, Level)

각 노드가 포함된 노드의 타입을 '|'을 이용하여 표현하였으며 이를 통해 타입 정보를 내장하고 있는 노드들의 타입 정보를 함께 컨텍스트에 추가함으로써 각 컨텍스트가 담고 있는 정보를 더 고도화 될 수 있다. 즉, setLevel 함수의 코드가 입력으로 들어왔을 때 함수의 역할인 Speaking 변수의 키(Student)와 값(Level)을 추가하는 작업을 한다는 것을 예측해야하며 이는 기존 타입 정보가 없는 경로보다 각 노드의 타입 정보를 추가한 컨텍스트를 이용하여 함수의 특징을 더 잡을 수 있다.

이러한 특징은 다양한 소스코드 중 타입 정보에 영향을 많이 받는 소스코드일수록 더 좋은 결과를 낼 수 있다. 이와 같은 결과는 본 논문 4장에서 기존 연구와 비교함으로써 평가한다.

```java
package Students;

import java.util.HashMap;

public class English {
    private HashMap<Student, String> Speaking = new HashMap<>();
    public void setLevel(Student student, String Level) {
        Speaking.put(student, Level);
    }
}
```

(a) Sample source code



(b) Add type Information

그림 12 Javaparser를 이용한 타입 정보 추출 예제

2) 타입 정보 추출 과정

[그림 11]의 2번과 같이 데이터 전처리의 첫 단계는 파싱을 통해 각 소스코드의 추상 구문 트리를 생성하고 모든 식별자의 타입 정보를 저장하는 것이다. 자바환경에서 추상 구문 트리는 Javaparser를 이용하여 생성할 수 있으며 Javaparser의 VoidVisitorAdapter를 상속받아 파서의 Visitor를 조작할 수 있다. 우리는 Visitor를 이용하여 클래스 내의 모든 식별자의 타입 정보를 추출하여 저장하고 3번 과정인 경로를 추출할 때 저장된 식별자의 타입을 이용하여 컨텍스트에 타입 정보를 추가한다.

일단 식별자의 타입 정보를 추출해야한다. 모든 식별자의 타입 정보를 추출하기 위해 파싱 과정동안 변수가 선언되면 각 변수에 타입을 맵핑하여 저장해야한다. 자바의 경우 선언문은 필드변수와 지역변수에서 나올 수 있다. 보통의 Visitor의 경우 파싱은 트리의 루트에서 시작하여 자식노드로 방문하여 추상 구문 트리를 형성한다. 즉, 모든 소스코드는 코드의 루트 노드인 CompilationUnit를 시작하여 모든 자식노드가 단말 노드에 도착할 때까지 탐방을 하며 추상 구문 트리를 생성한다.

따라서 우리는 소스코드 내의 모든 필드변수와 지역변수를 저장하기 위해서 Visitor가 클래스 선언 노드를 만나면 필드변수의 타입 정보를 저장하고 그 이후 각 블록마다 사용되는 지역변수 및 함수의 각 파라미터 타입 정보를 저장한다. 여기서 핵심 아이디어는 각각의 클래스와 블록은 고유의 번호가 부여가 되며 경로 추출 과정에서 타입 정보가 필요한 노드를 만났을 때 해당 노드가 속해 있는 블록에서 타입 정보를 찾는 것이다.

또한 필드변수와 지역변수를 관리하기 위해 소스코드의 각 클래스별로 클래스 내의 변수들을 관리할 객체가 필요하다. 즉, [그림 12]의 경우 우리는 English 클래스를 파싱할 때 English 클래스 내에 Speaking 변수의 타입이 HashMap이라는 정보를 저장할 객체가 필요하다. 우리는 이와 같이 클래스 내의 식별자 정보를 관리하기 위해 아래와 같이 두 개의 변수를 갖는 ClassContent 클래스를 정의하여 타입 정보들을 관리한다.

```
private Map<String, String> c_Field_Types = new HashMap<String, String>();
private ArrayList<Map> c_Block_Types = new ArrayList<>();
```

여기서 c_Field_Types 변수는 필드변수의 이름과 타입 정보가 저장되는 Map이다. 즉, English 클래스의 경우 c_Field_Types는 [Speaking : HashMap<Student, String>]이 저장된다. 그리고 c_Block_Types 변수는 클래스 내의 존재하는 모든 블록의 지역변수 정보를 저장하는 ArrayList이다. 클래스 내부엔 여러 개의 블록이 존재할 수 있으며 각각의 블록마다 지역변수를 구분하여 사용한다. 즉, 클래스 안에 두 개의 함수가 있다면 각 함수에서 선언된 동일한 이름의 변수는 타입이 다를 수 있으며 이를 구분하여 저장해야한다.

[그림 13]은 파싱과정에서 필드변수와 지역변수의 타입 정보를 저장하고 사용하는 과정의 예를 보여준다. 그림은 임의의 코드의 AST 일부분을 나타낸 것이며 이를 통하여 타입 정보를 추출하는 과정을 3단계로 나누어 설명한다.

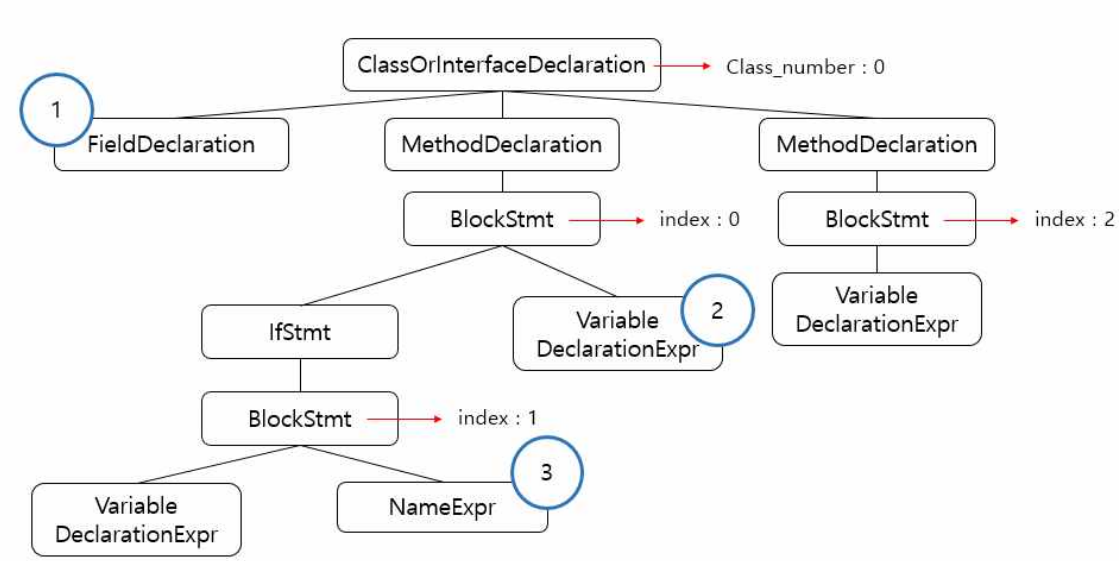


그림 13 타입 정보를 추출하는 과정을 보여주는 AST

가) 필드변수 타입 저장

첫 번째 단계는 클래스 내의 식별자 정보를 갖는 ClassContent 인스턴스를 생성하는 것이다. [그림 13]과 같이 Visitor가 파싱 하는 과정에서 클래스 선언 노드(ClassOrInterfaceDeclaration)를 방문하였을 때 해당 클래스의 번호를 부여하고 ClassContent 인스턴스를 생성한다. 이 인스턴스는 해당 클래스가 갖고 있는 모든 변

수를 관리하며 이를 위해 가장 먼저 ClassOrInterfaceDeclaration의 자식노드인 FieldDeclaration 노드를 이용하여 클래스가 갖고 있는 필드변수들의 이름과 타입을 추출한다.

FieldDeclaration 노드는 해당 클래스가 갖고 있는 필드변수의 정보를 갖고 있으며 해당 클래스가 갖고 있는 FieldDeclaration들을 이용하여 필드변수의 이름과 타입을 추출한 후 해당 클래스 ClassContent 인스턴스의 c_Field_Types 변수에 맵핑하여 저장한다.

나) 지역변수 타입 저장

해당 클래스의 필드변수를 모두 저장하였다면 다음은 c_Block_Types인 모든 지역변수를 저장하는 것이다. 지역변수는 선언된 각 블록 내에서만 접근 가능한 변수를 뜻한다. 따라서 각 변수가 속해있는 블록을 기억하기 위해 각 블록에 번호를 부여하여 관리해야하며 이를 위해 파싱 과정 중 BlockStmt 노드를 만날 때마다 해당 블록에 번호를 부여하고 블록이 갖고 있는 변수 정보를 맵핑하여 Map을 만들어 c_Block_Types 변수에 저장하여야한다.

즉, [그림 13]과 같이 각 BlockStmt에는 index 값이 부여가 되며 각 블록 안에서 선언된 변수는 VariableDedclaration을 이용하여 추출할 수 있다. 또한 그림에는 없지만 함수의 경우 파라메타의 경우 역시 지역변수를 나타내기 때문에 Parameter 노드를 이용하여 파라메타 변수 정보를 추출해야한다.

여기서 주의해야 할 점은 각 블록이 소유하고 있는 변수는 해당 블록 노드의 자식 노드들만 저장한다. 따라서 그림에서 0번 블록의 경우 1번 블록이 갖고 있는 VariableDedclaration가 갖고 있는 변수는 관여하지 않는다. 이는 각 블록이 소유하고 있는 변수를 좀 더 확실하게 구분하여 사용하기 위함이다.

이처럼 VariableDedclaration를 이용하여 각 블록이 갖고 있는 변수의 정보를 Map으로 저장했다면 각 블록이 현재 소속되어 있는 클래스의 번호를 이용하여 소속된 ClassContent 인스턴스를 찾고 해당 인스턴스가 갖고 있는 c_Block_Types에 각 블록의 변수 Map을 저장한다.

이로써 현재 클래스 내의 모든 필드변수와 지역변수의 변수의 타입 정보를 갖는 인스턴스를 만들었다. 여기까지가 [그림 11]의 2번 과정에 해당된다. 파싱을 통해 AST를 만들었으며 AST를 만드는 파싱 과정동안 해당 클래스가 갖고 있는 모든 변수의

타입 정보를 저장하였다. 이제 이를 이용하여 모든 경로를 구할 때 저장한 타입 정보를 이용하여 타입 정보를 추가한 경로를 만들 수 있다.

다) 타입 정보 추출

경로를 추출하는 과정은 code2seq와 비슷하다. 각 터미널 노드에서 다른 터미널 노드까지 도달하는 모든 경로를 구하는 것이다. 다만 우리가 제안하는 데이터의 경우 각 경로를 구하는 과정에서 노드가 타입 정보를 갖고 있다면 경로에 타입 정보를 추가하여 컨텍스트를 만드는 것이다. 따라서 각 노드가 갖고 있는 타입 정보가 무엇인지 체크하는 과정이 필요하다.

우리는 파싱하는 과정에서 클래스 내의 모든 필드변수와 지역변수가 갖고 있는 타입을 저장해놓았다. 이를 이용하여 각 노드가 갖고 있는 타입을 체크해야한다. 또한 앞에서 설명했듯이 각 변수는 소속되어 있는 블록 한정에서 사용되어야한다. 이를 위해 타입 정보를 알고 싶은 노드를 만났을 때 해당 노드가 소속되어 있는 블록들의 인덱스 값을 알아야한다.

[그림 13]의 3번을 보면 NameExpr은 식별자의 이름을 나타내는 노드로 타입 정보를 갖고 있다. 따라서 타입 정보를 알기 위해 가장 먼저 해당 노드가 속해있는 블록들의 인덱스 값을 알아야하며 이를 알기 위해 루트 노드인 CompilationUnit에 도달할 때까지 부모노드를 방문하여 소속되어 있는 블록들을 체크한다. 즉, 3번의 NameExpr 노드의 경우 소속되어 있는 블록은 1번 블록과 0번 블록이며 0번 클래스에 소속되어 있는 것을 확인할 수 있다.

노드가 소속되어 있는 블록과 클래스를 알았다면 해당 클래스 인스턴스에서 타입을 찾아야한다. 즉, 3번 노드의 타입 정보를 알기 위해 노드가 속해있는 ClassContent 인스턴스에서 소속되어 있는 블록들의 번호를 이용하여 타입을 알아낼 수 있다. 가장 먼저 c_Block_Types의 index가 1인 Map에서 해당 식별자의 이름을 갖는 값을 찾는다. 만약 없다면 index가 0인 Map에서 찾으며 마찬가지로 없다면 필드변수에서 찾는다. 이처럼 타입 정보를 갖고 있는 노드는 경로를 구성하는 과정에서 타입을 알아낼 수 있으며 경로의 각 노드 다음 노드로 타입 정보를 입력하여 경로를 구성한다.

나. 딥러닝 모델 구조

본 연구는 code2seq[3]에서 사용한 모델을 사용한다. 본 절에서는 앞서 2장 나. 에서 설명한 모델에 대해 자세히 설명한다. 모델의 구조는 [그림 5]와 같으며 모델을 Encoder, Attention Part, Decoder로 구분하여 예제를 통해서 설명한다.

1) Encoder

딥러닝 모델의 입력부분은 [그림 14]와 같으며 [그림 5]의 앞부분에 해당한다. 2장에서 설명했듯이 입력으로 들어오는 데이터 형식 컨텍스트는 $(Token_s, Path, Token_e)$의 형태로 들어온다. 또한 하나의 함수코드당 다수의 컨텍스트가 있으며 이 컨텍스트들로 함수를 분석해야한다. [그림 14]는 다수의 컨텍스트들 중 하나의 컨텍스트가 인코딩되는 아키텍쳐를 보여준다. 즉, 각 컨텍스트마다 그림과 같이 인코딩이 되며 인코딩된 컨텍스트들을 이용하여 코드의 특징을 추출한다. 인코더의 역할은 데이터를 하나의 벡터로 압축시키는 역할을 한다. 따라서 우리가 사용한 모델의 경우 하나의 컨텍스트가 의미가 있는 하나의 벡터로 압축되어야한다. 그림을 보면 하나의 컨텍스트가 하나의 벡터로 압축되는 과정을 볼 수 있으며 각각의 시작토큰, 경로, 끝토큰이 하나의 벡터로 압축되는 과정을 설명한다.

가) 임베딩(Embedding)

컨텍스트의 모든 입력은 임베딩(Embedding)과정이 필요하다. 임베딩이란 단어를 하나의 벡터로 표현하는 것을 뜻한다. 딥러닝 모델은 Speaking, Nm과 같은 단어를 이해하지 못한다. 따라서 이러한 단어를 하나의 벡터로 변환시켜야하며 이를 위해 임베딩기술이 사용된다. 임베딩의 특징은 단순히 단어를 벡터로 변환시키는 것이 아닌 각 단어들의 연관성에 따라서 벡터를 구성하며 비슷한 성질을 갖는 단어들은 유사한 벡터 값으로 변환된다. 단어 x의 임베딩 벡터 E_x는 훈련을 통해서 학습이 되며 아래와 같다.

$$E_x = Embedding(x)$$

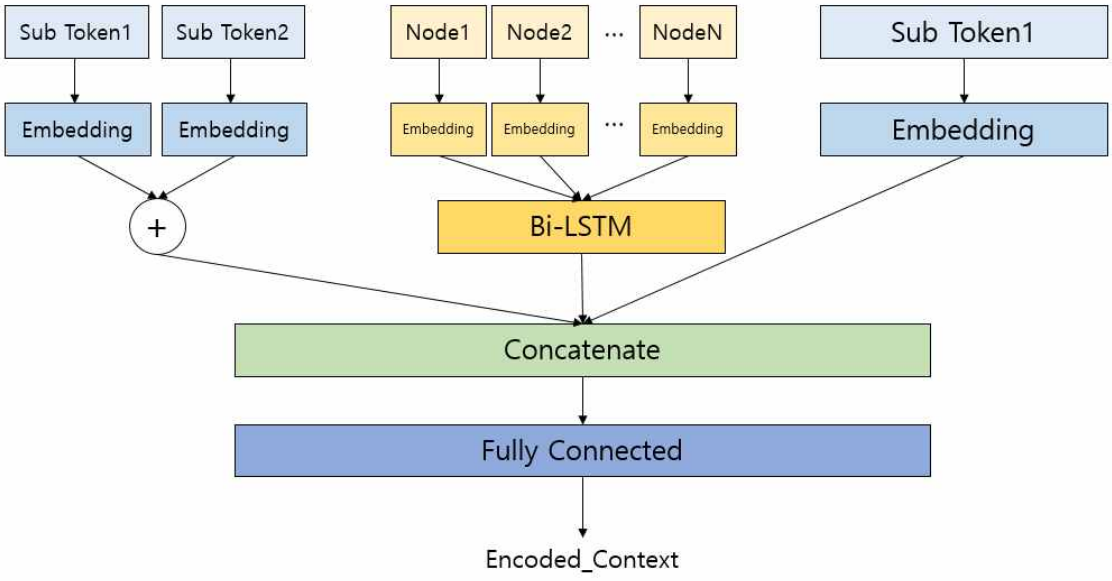


그림 14 모델의 인코더

나) 토큰(Token)

각 토큰은 다수의 서브 토큰으로 분리될 수 있다. 예를 들어, 시작 토큰으로 StudentNum이라는 단어가 들어왔다고 가정하자. 이 토큰은 Student와 Num의 서브토큰으로 분리가 된다. 이는 [17]에서 자연어 처리에 있어 서브 단어화는 그 단어의 특징을 추출하는데 더욱 유리하다는 연구 결과를 기반으로 한다. 예를 들어, '물안경'의 경우 '물'과 '안경'을 분리시킴으로써 물안경이 물과 안경의 특징을 모두 갖고 있다는 특징을 추출할 수 있다. 이를 바탕으로 N개의 서브토큰으로 토큰의 임베딩 값은 아래와 같다. 서브토큰의 임베딩 값을 모두 더함으로써 단어의 의미를 더욱 세밀화 된다.

$$E_{Token} = \sum_{n=1}^{N} E_{Sub-Token(n)}$$

다) 경로(Path)

인코더는 다수의 노드와 타입 정보의 시퀀스로 구성된 경로를 하나의 의미 있는 벡터로 압축하고 여기에 토큰 임베딩 값을 추가함으로써 컨텍스트를 압축한다. 따라서 경로 시퀀스의 특징을 잘 추출하여 벡터로 압축하여야한다. 경로는 시작 토큰부터 끝

토큰까지 도달하는데 모든 노드와 타입 정보의 시퀀스로 구성되어 있다. 각각의 노드와 타입은 임베딩되어 벡터로 표현되며 시퀀스데이터의 특징을 추출하기 위해 LSTM을 사용해야한다. 하지만 LSTM의 정보 소실문제를 해결하기 위해 양방향 LSTM 구조로 된 Bi-LSTM을 사용하며 이를 통해 시퀀스의 초반 데이터가 소실하는 문제를 완화할 수 있다. 따라서 인코딩된 경로는 아래와 같다. 이를 통해 다수의 노드와 타입으로 구성된 경로는 하나의 이미를 갖는 벡터로 압축되며 이를 통해 경로의 특징을 추출할 수 있다.

$$V_{Path} = Bi-LSTM(E_{Node1}, E_{Node2}, \cdots E_{NodeN})$$

이와 같이 $(Token_s, Path, Token_e)$의 형태로 들어온 하나의 컨텍스트 데이터는 임베딩과 인코딩을 통해 각각의 의미 있는 벡터로 표현되며 이를 아래와 같이 연쇄시켜 하나의 벡터로 만들어 특징이 추출되며 이를 Fully Connected Layer(Dense Layer)를 통해 다시 한 번 특징이 강조된다.

$$Encoded\_Context(Context_n) = Dense(Concatenate(E_{Token_s}, V_{Path}, E_{Token_e}))$$

즉, 예를 들어 (Speaking, NM | HashMap<Student, String> -> ... -> VDID | String, Level)의 컨텍스트 데이터는 인코더를 거쳐 (6.1, 2.5, 4.5, 1.5, ...)와 같은 의미가 압축된 벡터로 표현되며 이러한 컨텍스트 벡터들을 모아 코드 특징 추출을 한다.

2) Decoder

입력데이터는 하나의 함수마다 다수의 컨텍스트로 이루어진 컨텍스트 모음으로 구성되어 있다. 앞의 2장 나. 에서 설명했듯이 총 n개의 단말 노드를 갖고 있는 하나의 함수는 함수 라벨링을 위해 ${}_nC_2$개의 컨텍스트를 이용한다. 따라서 하나의 함수마다 1)에서 인코딩된 컨텍스트 벡터가 디코더로 ${}_nC_2$개 들어오며 이를 통해 시퀀스 형태의 함수 라벨링을 진행한다. 여기서 중요한 것은 디코더에서 결과를 도출할 때 각각의 컨텍스트마다 중요도가 다를 것이며 이를 위해 Attention Mechanism을 사용한다.

여기서 사용되는 디코더는 2장 다. 에서 설명한 Encoder-Decoder 및 Attention Mechanism과 원리는 같지만 약간 다르게 작동된다. [그림 15]는 모델의 디코더

구조이며 [그림 16]은 디코더 구조에서 Attention Mechanism을 이용하여 시퀀스를 예측하는 구조를 도식화한 것이다. 기존의 Encoder-Decoder 구조와 다르게 각 입력 데이터가 독립적인 컨텍스트 벡터이다. 여기서 N개의 컨텍스트 벡터는 앞의 인코더를 통해서 인코딩된 ${}_nC_2$개의 컨텍스트 벡터를 뜻한다. 즉 디코더는 인코딩된 N개의 컨텍스트 벡터를 시퀀스의 각 출력마다 Attention Layer를 통해 예측하는데 사용한다. 따라서 디코더의 출력은 아래와 같다.

$$y_i = LSTM(Attention(Encoded\_Context1, \ldots, Encoded\_ContextN))$$

따라서 [그림 15]와 같이 N개의 컨텍스트는 시퀀스의 출력을 순차적으로 만들어 낸다. 각각의 출력은 가장 중요시하게 보는 컨텍스트가 다르며 훈련을 통해 Attention Weight를 학습하고 그림과 같이 컨텍스트별로 중요도 가중치를 두어 예측한다.

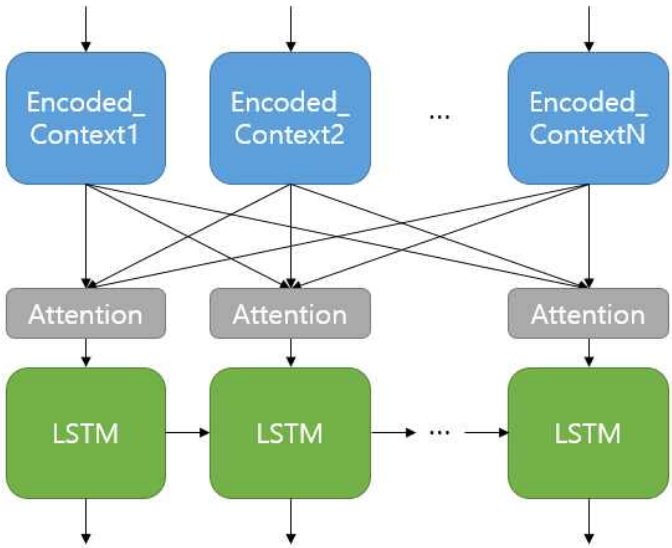


그림 15 타입 기반 소스코드 특징 추출 모델의 디코더

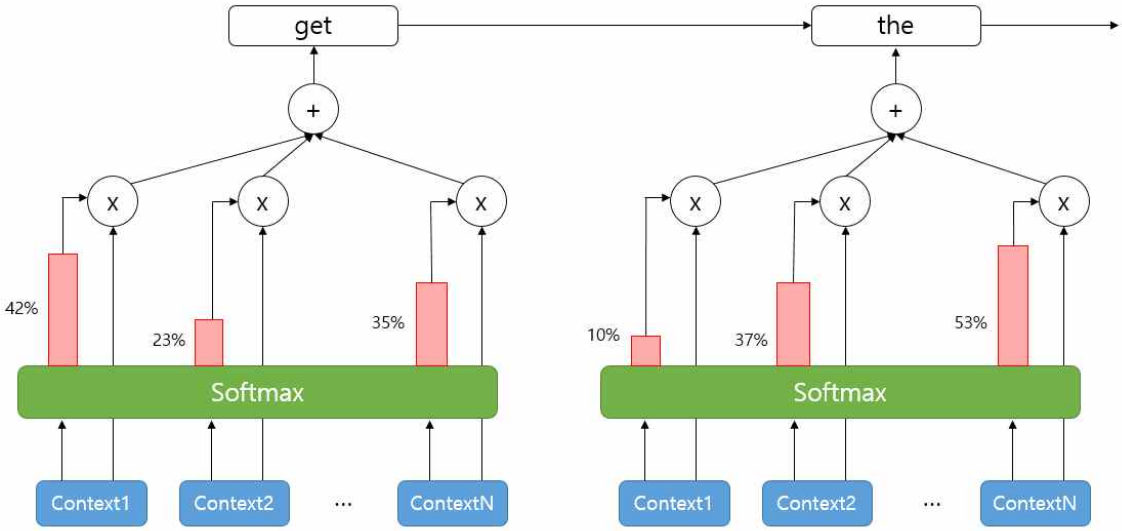


그림 16 타입 기반 소스코드 특징 추출 모델의 Attention Mechanism

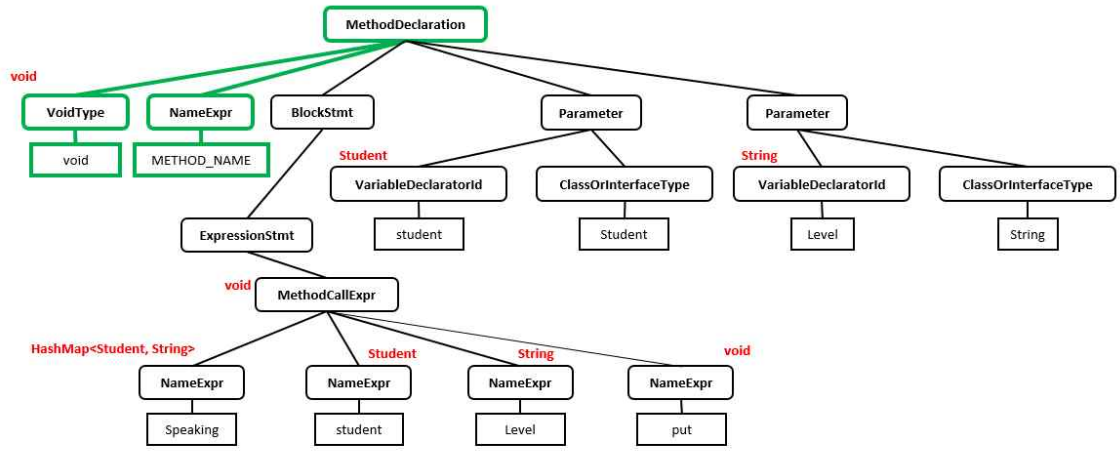
다. 수행 작업

1) 함수명 예측 작업

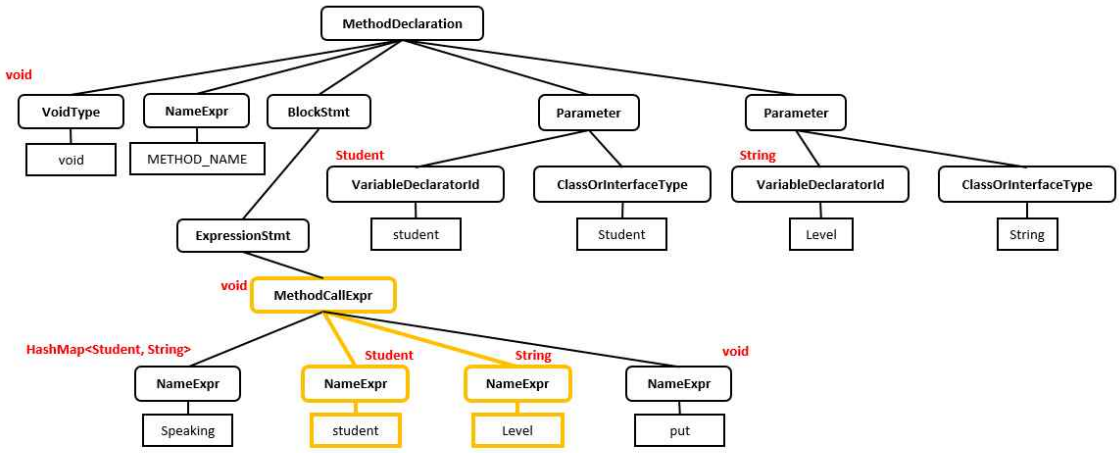
타입 기반 소스코드 특징 추출 모델을 이용하여 우리는 두 가지의 작업을 수행한다. 첫 번째 작업은 함수명 예측이다. 이는 임의의 함수 소스코드를 보고 함수명을 예측하는 것이다. [그림 17]은 앞에서 사용한 setLevel 함수의 이름을 METHOD_NAME라고 치환하고 setLevel이라는 함수명을 예측하는 작업을 보여준다. 경로의 각 노드 구분은 '|'를 이용해 분리하여 학습의 데이터로 사용되며 그림과 같이 각 서브토큰별로 다른 어텐션 스코어를 만들어 사용한다. 즉, 'set'을 예측할 때 모든 컨텍스트를 이용하지만 'void,Void|void|Mth|Nm,METHOD_NAME' 컨텍스트는 약 4.7%의 가중치를 두고 'level,Nm|String|Cal|Nm,put' 컨텍스트는 약 4.4%의 가중치를 두어 학습에 이용한다. 따라서 모든 컨텍스트의 가중치를 모두 합하면 100%가 되는 어텐션 가중치를 이용하여 각 서브토큰을 예측할 수 있다.

이러한 딥러닝 기반 소스코드 특징 추출 모델을 함수명 예측 작업에 사용함으로써 얻을 수 있는 몇 가지의 장점이 존재한다. 첫째로 대규모 프로젝트에서 함수명 규칙을 통일화 할 수 있다. 함수명 규칙 통일화는 코드의 가독성을 향상시키며 프로젝트의 질을 향상시킬 수 있다. 두 번째 장점은 개발 속도를 향상시킬 수 있다. 초보 개발자의 경우 함수명 명명과정을 어려워하는 개발자가 존재한다. 이러한 개

발자들을 위해 자동으로 함수명을 지정해줌으로써 개발자들이 함수명을 고민하는 일을 줄일 수가 있어 개발 속도를 향상시킬 수 있다. 이 외에도 다양한 장점이 있으며 나아가 높은 성능의 함수명 예측 작업은 다양한 분야에 활용될 수 있다.



Context	Attention Score	
void,Void|void|Mth|Nm,METHOD_NAME	0.047050	**set**
level,Nm|String|Cal|Nm,put	0.044318	
string,Cls|Prm|String|Mth|Bk|Ex|Cal|Nm,put	0.038406	
student,Nm|Student|Cal|Nm,put	0.037269	



Context	Attention Score	
student,Nm|Student|Cal|Nm|String,level	0.110052	**Level**
student,Cls|Prm|Student|Mth|Bk|Ex|Cal|Nm|String,level	0.083625	
level,VDID|Prm|String|Mth|Bk|Ex|Cal|Nm|String,level	0.069856	
student,VDID|Prm|Student|Mth|Bk|Ex|Cal|Nm|String,level	0.052171	

그림 17 함수명 예측 작업

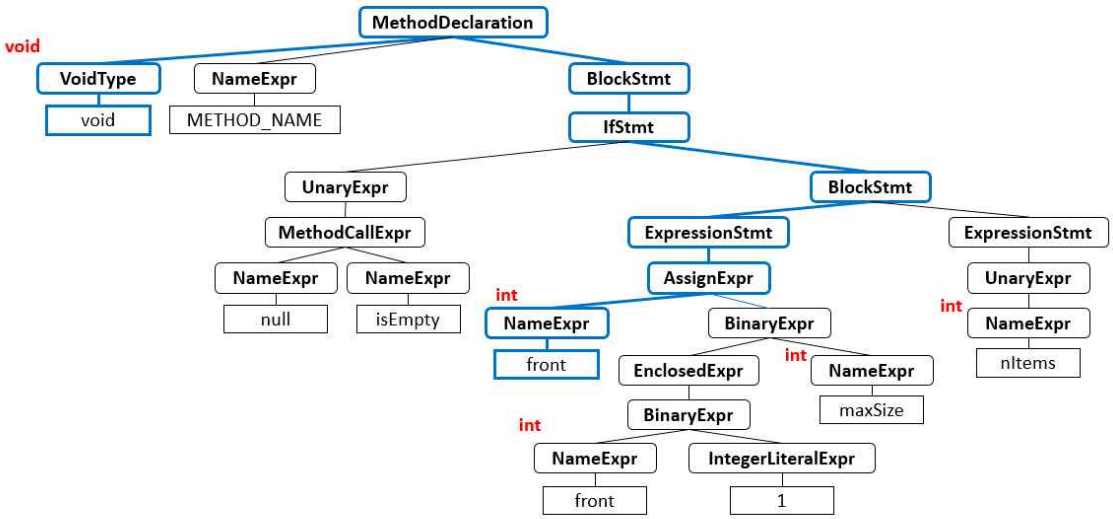
2) 함수 요약 작업

타입 기반 소스코드 특징 추출 모델을 이용한 두 번째 작업은 함수 요약 작업이다. 이 작업은 함수의 코드 내용을 보고 한 줄로 요약하는 것이다. [그림 19]는 예제소스 [그림 18]의 remove 함수를 요약하는 작업을 보여준다. [그림 18]의 코드는 일반적인 큐 객체의 코드 중 일부이며 remove 함수는 큐의 첫 번째 아이템을 제거하는 역할을 하는 함수이다. 우리는 주어진 필드 타입 정보와 remove 함수의 코드를 보고 이를 한 줄로 요약하는 작업을 수행할 수 있어야한다. 작업 과정은 함수명 예측과 유사하다. 출력 시퀀스의 각 단어는 각각 다른 어텐션 스코어를 이용하여 하나의 문장을 예측한다. 디코더를 이용하여 시퀀스를 만들어내기 때문에 자연스러운 하나의 문장을 만들어 낼 수 있다. 즉, [그림 19]처럼 Queue 객체의 remove 함수를 'Removes the last item from the queue.'로 예상할 수 있다.

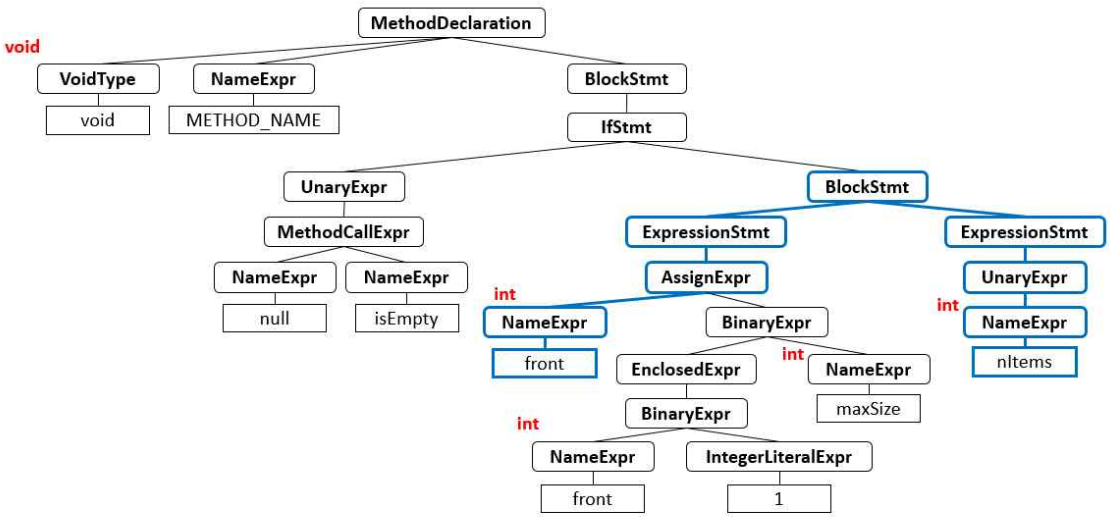
이러한 함수 요약 작업은 다양한 분야에 활용될 수 있다. 특히 오픈소스 시장에서 효율적으로 활용될 수 있을 것이라고 예측한다. 최근 오픈소스 시장이 커지고 있으며 오픈소스 저장소에는 대규모의 소스들이 저장되어 있다. 오픈소스를 활용하는 개발자에게 각 함수의 요약된 정보를 제공해 준다면 개발자들은 코드를 분석하는 시간을 아낄 수 있다. 즉, 각 함수에 함수 내용을 요약하는 주석을 추가할 수 있으며 이는 오픈소스를 제공하는 개발자 입장에서도 시간을 효율적으로 활용할 수 있다. 함수 요약 작업을 통해 함수 내용을 자동으로 요약할 수 있기 때문에 개발자는 각 함수의 주석을 따로 작성할 필요가 없으며 이는 개발 속도를 향상시킬 수 있는 장점으로 다가갈 수 있기 때문이다. 이처럼 함수 내용 요약 작업은 개발자에게 개발 속도를 향상시키고 효율적인 프로그래밍을 가능하게 할 것이다.

```
class Queue {
    private int maxSize;
    private int[] queueArray;
    private int front;
    private int rear;
    private int nItems;

    public void remove() {
        if (!isEmpty()) {
            front = (front + 1) % maxSize;
            nItems--;
        }
    }

            ⋮

}
```

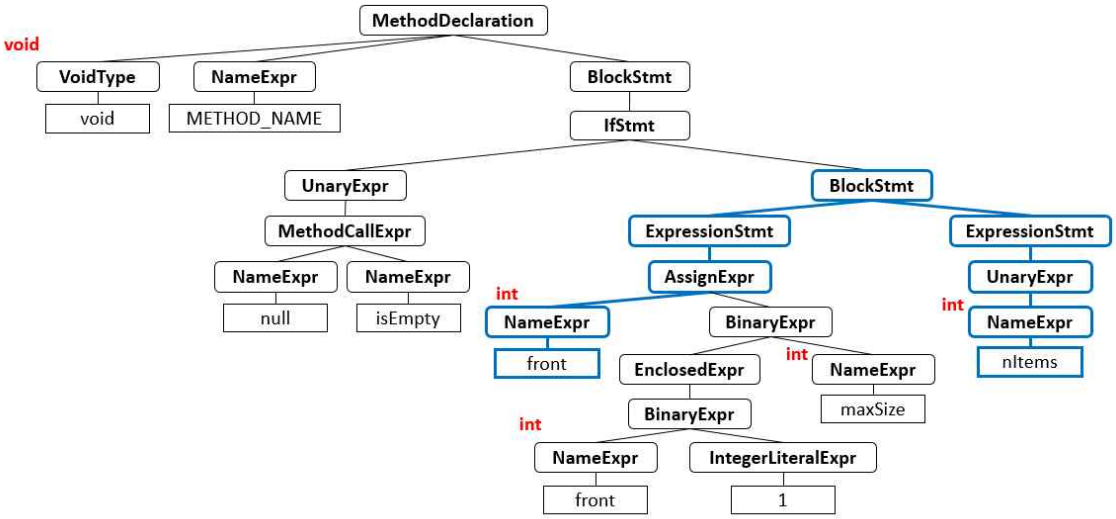
그림 18 함수 요약 작업 예제 코드



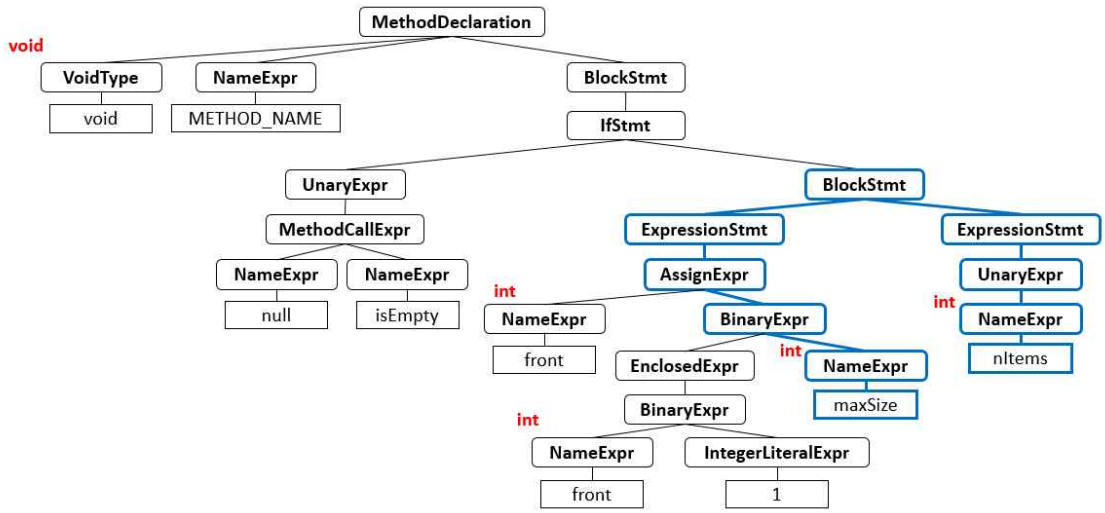
Context	Attention Score	
void,Void|void|Mth|Bk|If|Bk|Ex|As|Nm|int,front	0.041753	**Remove**
METHOD_NAME,Nm|Mth|Bk|If|Bk|Ex|As|Nm|int,front	0.027235	
void,Void|void|Mth|Bk|If|Bk|Ex|PosDec|Nm|int,n|items	0.024464	
METHOD_NAME,Nm|Mth|Bk|If|Bk|Ex|PosDec|Nm|int,n|items	0.021591	



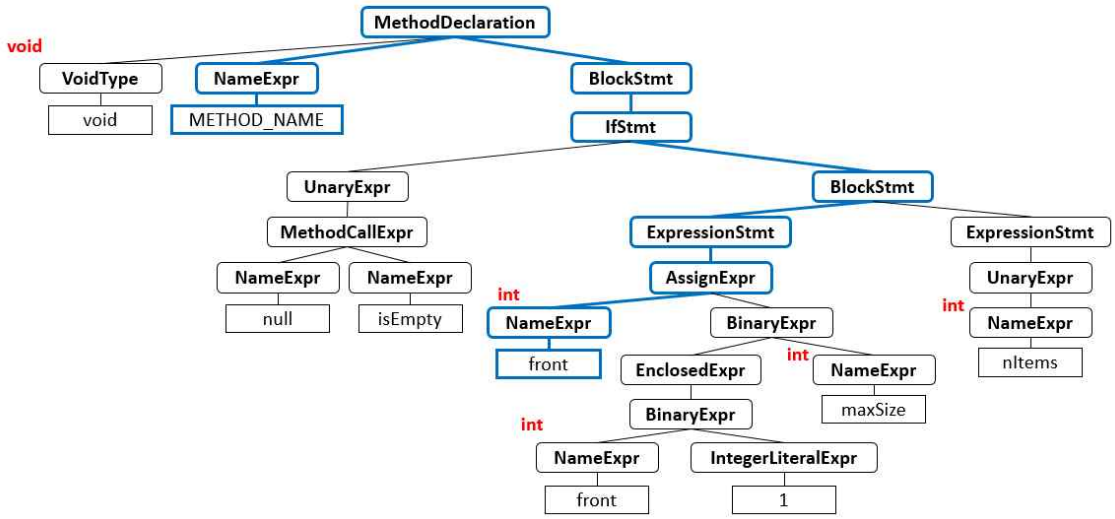
Context	Attention Score	
front,Nm|int|As|Ex|Bk|Ex|PosDec|Nm|int,n|items	0.037413	**the**
METHOD_NAME,Nm|Mth|Bk|If|Bk|Ex|As|Nm|int,front	0.029181	
METHOD_NAME,Nm|Mth|Bk|If|Bk|Ex|PosDec|Nm|int,n|it	0.025167	
max|size,Nm|int|Mod|As|Ex|Bk|Ex|PosDec|Nm|int,n|items	0.023090	



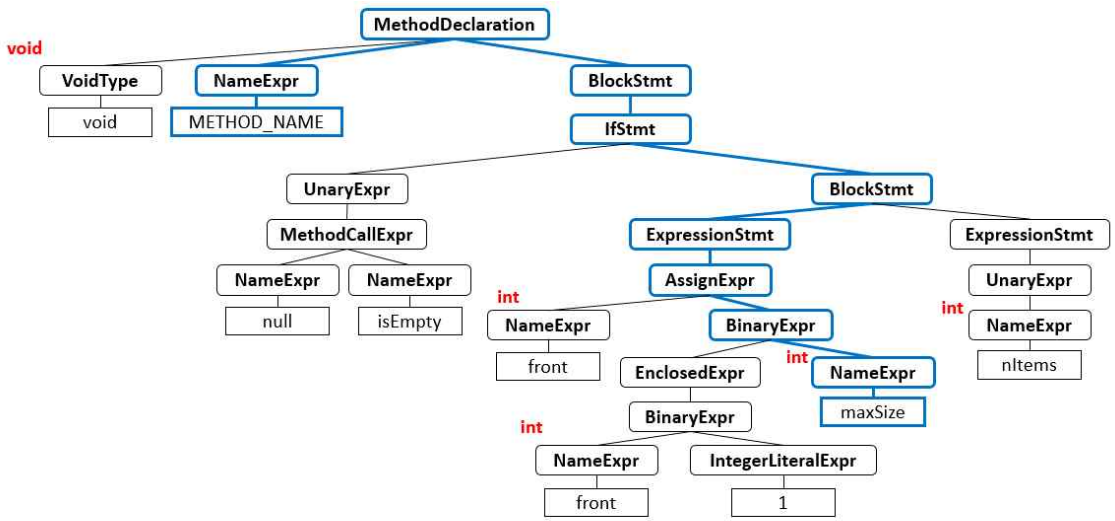
Context	Attention Score	
front,Nm|int|As|Ex|Bk|Ex|PosDec|Nm|int,n|items	0.101673	**last**
max|size,Nm|int|Mod|As|Ex|Bk|Ex|PosDec|Nm|int,n|items	0.097954	
METHOD_NAME,Nm|Mth|Bk|If|Bk|Ex|As|Mod|Nm|int,max|size	0.082053	
front,Nm|int|As|Mod|Nm|int,max|size	0.077575	



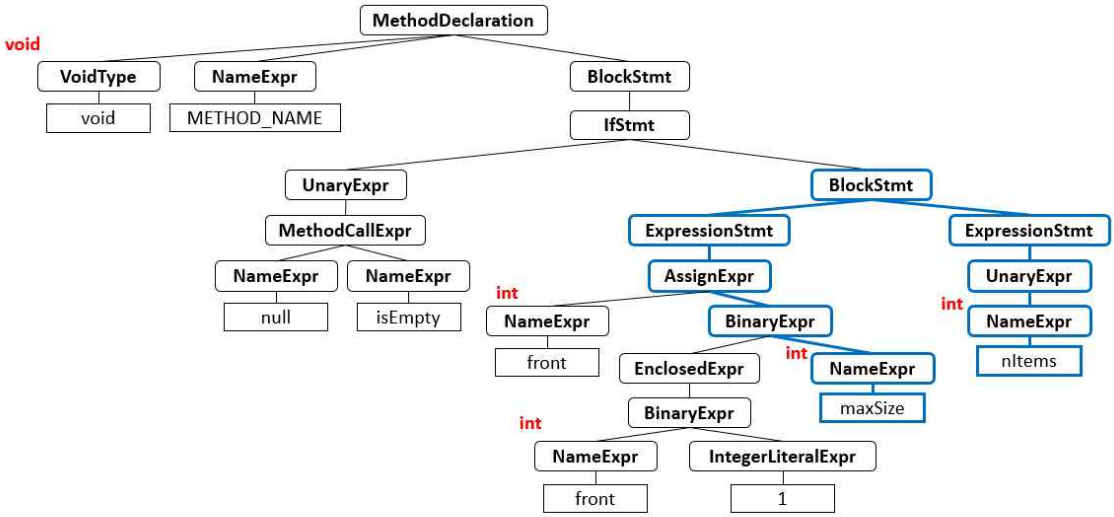
Context	Attention Score	
max|size,Nm|int|Mod|As|Ex|Bk|Ex|PosDec|Nm|int,n|items	0.133323	**item**
front,Nm|int|As|Ex|Bk|Ex|PosDec|Nm|int,n|items	0.099694	
METHOD_NAME,Nm|Mth|Bk|If|Bk|Ex|As|Mod|Nm|int,max|size	0.079569	
METHOD_NAME,Nm|Mth|Bk|If|Bk|Ex|PosDec|Nm|int,n|items	0.071814	



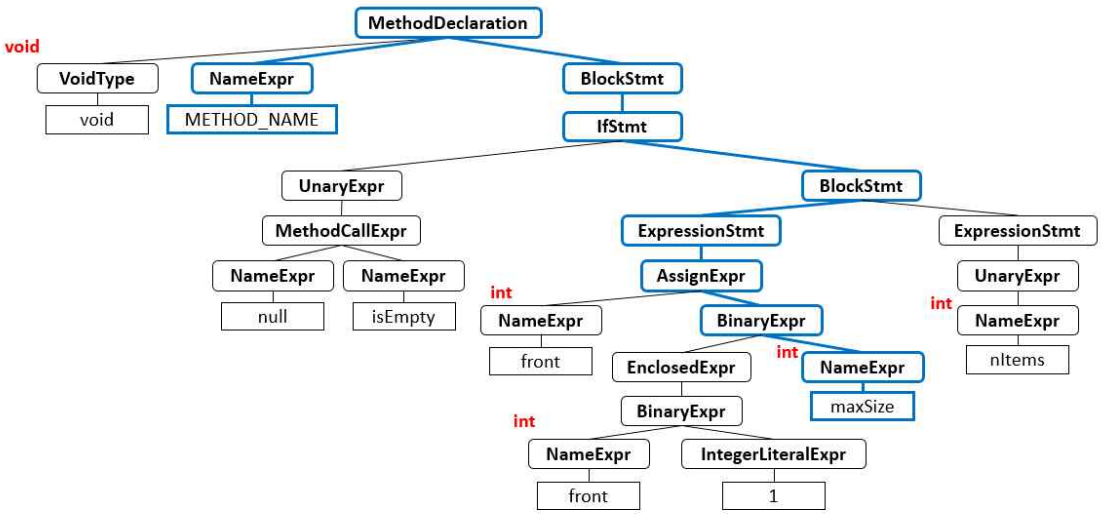
Context	Attention Score	
METHOD_NAME,Nm|Mth|Bk|If|Bk|Ex|As|Nm|int,front	0.086571	**from**
void,Void|void|Mth|Bk|If|Bk|Ex|As|Nm|int,front	0.076549	
front,Nm|int|As|Mod|Enc|Plus|Nm|int,front	0.046476	
METHOD_NAME,Nm|Mth|Bk|If|Bk|Ex|As|Mod|Nm|int,max|size	0.046024	



Context	Attention Score	
METHOD_NAME,Nm|Mth|Bk|If|Bk|Ex|As|Mod|Nm|int,max|size	0.053852	**the**
is|empty,Nm|Cal|Not|If|Bk|Ex|As|Mod|Nm|int,max|size	0.048229	
void,Void0|void|Mth|Bk|If|Bk|Ex|As|Mod|Nm|int,max|size	0.046470	
max|size,Nm|int|Mod|As|Ex|Bk|Ex|PosDec|Nm|int,n|items	0.043202	



Context	Attention Score	
max|size,Nm|int|Mod|As|Ex|Bk|Ex|PosDec|Nm|int,n|items	0.171258	**queue**
front,Nm|int|As|Ex|Bk|Ex|PosDec|Nm|int,n|items	0.140560	
METHOD_NAME,Nm|Mth|Bk|If|Bk|Ex|PosDec|Nm|int,n|items	0.093595	
front,Nm|int|As|Mod|Enc|Plus|Nm|int,front	0.066683	



Context	Attention Score	
METHOD_NAME,Nm|Mth|Bk|If|Bk|Ex|As|Mod|Nm|int,max|size	0.023319	.
is|empty,Nm|Cal|Not|If|Bk|Ex|As|Mod|Nm|int,max|size	0.021958	
front,Nm0|int|As|Ex|Bk|Ex|PosDec|Nm|int,n|items	0.020444	
METHOD_NAME,Nm|Mth|Bk|If|Bk|Ex|PosDec|Nm|int,n|items	0.019937	

Result : Remove the last item from the queue.

그림 19 함수 요약 작업

4. 결과분석 및 논의사항

가. 평가

1) 데이터 셋

우리는 앞에서 소개한 두 가지 작업에 대하여 다른 데이터 셋을 사용한다. 각 작업에서 사용한 데이터 셋은 작업의 방식에 따라 다른 형식의 데이터 셋을 사용해야 한다. 즉, 함수명 예측 작업의 경우 함수 소스코드를 입력으로 받아 함수명을 출력을 내야하며 함수 요약 작업의 경우 함수 소스코드를 입력으로 받아 한 줄의 요약된 문장을 출력으로 내야한다.

가) 함수명 예측 데이터 셋

우리는 함수명 예측을 위해 [3]에서 제공하는 데이터 셋을 사용했다. 함수명 예측 작업의 경우 함수 소스코드와 함수명을 이용하여 학습을 진행해야한다. 따라서 데이터 셋의 형식은 소스코드와 함수명이 모두 제공되는 데이터 셋이어야 한다. [3]에서 제공하는 대규모 소스코드 데이터 셋은 Github에 올라온 다양한 레파지토리에서 Java 소스코드들만을 수집한 형식으로 구성되어 있다. 레파지토리에 등록된 Java 소스코드의 모든 함수는 함수명을 갖고 있기 때문에 데이터 전처리를 통해서 각 함수의 소스코드와 함수명을 분리하여 사용한다.

데이터 셋은 데이트의 크기에 따라 small, medium, large 크기의 데이터 셋을 제공한다. small 데이터 셋의 경우 약 660MB의 소스코드로 구성되어 있으며 약 11만개의 함수를 갖고 있다. 또한 large 데이터 셋의 경우 약 15GB의 소스코드로 구성되어 있으며 약 1400만개의 함수를 갖고 있다. 우리는 이 데이터 셋 중 small 데이터 셋과 medium 데이터 셋을 사용한다.

나) 함수 요약 데이터 셋

함수 요약 작업의 경우 함수 소스코드와 함수의 내용을 요약하는 한 줄의 요약 문장을 이용하여 학습을 진행해야한다. 따라서 데이터 셋의 형식은 소스코드와 요약 문장을 모두 제공하는 데이터 셋이어야 한다. 우리는 함수 요약 작업을 위해 [4]에서 제공하는 CONCODE 데이터셋을 사용하였다. CONCODE 데이터 셋은 많은 개발자들

이 JavaDoc 스타일의 태그 형식으로 주석을 작성하는 방식을 이용하여 데이터 셋을 수집하였다. JavaDoc이란 Java 코드에서 API 문서를 HTML 형식으로 생성해주는 도구이다. [표 2]는 JavaDoc 스타일의 주석 형식을 보여준다. 표와 같이 주석은 '/**' 와 '*/'를 이용하여 형성되며 주석은 함수의 내용을 요약한 문장과 '@param'과 같은 다양한 태그로 구성되어 있다. CONCODE 데이터 셋은 Github에서 Java 소스 중 JavaDoc 스타일의 태그가 포함된 코드들을 수집하여 데이터 셋을 구성하였다. 태그가 포함된 함수의 경우 각 함수의 소스코드와 요약된 문장, 그리고 변수들의 타입 정보로 구성되어 있으며 약 10만개의 함수 정보를 갖고 있다. 우리는 이 데이터셋을 전처리를 통하여 함수의 소스코드를 다수의 컨텍스트로 구성된 데이터 형식으로 변형하여 데이터로 사용한다. 즉, 하나의 함수는 다수의 컨텍스트와 한 줄의 요약된 문장으로 구성된 데이터로 학습을 진행한다.

표 2 JavaDoc 예제

```
/**
 * A function that does something
 *
 * @param variable Description
 * @return Description
 */
public int methodName (...) {
    // method body with a return statement
}
```

2) 실험 환경

딥러닝 모델은 컴퓨팅 환경에 영향을 많이 미친다. 특히 GPU의 성장에 따른 딥러닝의 성장을 보이고 있는 만큼 GPU 환경이 딥러닝 모델에 큰 영향을 미친다는 것을 알 수 있다. 우리는 데이터 셋의 크기가 크고 모델이 복잡한 만큼 높은 사양의 컴퓨팅 환경에서 학습을 진행하였다. 우리가 사용한 GPU는 NVIDIA사의 Tesla V100이다. V100은 640개의 Tensor 코어를 탑재하여 100테라플롭스(TFLOPS) 수준의 성능을 보이며 32GB의 메모리를 사용하기 때문에 CPU 서버보다 47배 향상된 추론 성능을 보인다. 이는 약 8테라플롭스 수준의 성능을 보이는 NVIDIA사의 Geforce GTX 1080보다 약 6배 빠른 속도와 더 높은 메모리 수준에 따라 더 높은 추론 성능을 보인다. 또한 추가적으로 데이터 셋 크기가 크기 때문에 여유로운 학습을 위해 최소 1TB

의 저장 공간이 필요하다.

3) 결과

본 연구는 기존의 연구에서 타입 정보를 추가함으로써 향상되는 딥러닝 모델의 성능을 보이기 위해 진행이 되었다. 따라서 우리는 아래의 두 개의 방법을 통하여 결과를 판단한다.

- 대표적인 딥러닝 모델 성능지표인 F1 점수를 이용한 기존 모델과의 비교
- 기존 데이터 형식에 타입 정보를 추가했을 때 더 정확한 결과를 예측하는 예제를 통한 결과 비교

가) 성능 평가

우리는 모델의 성능을 기존의 [3] 연구의 결과와 비교하여 평가한다. 딥러닝 모델의 성능을 비교하기 위해 다양한 성능 지표가 있으며 우리는 그 중 데이터 출력이 불균형 구조일 때, 모델의 성능을 정확하게 평가할 수 있는 F1 점수를 사용한다. [그림 20]은 [3]연구에서 제공하는 데이터 셋을 [3]의 데이터 형식을 이용하여 학습한 모델의 F1 점수와 우리가 제안하는 데이터 형식을 이용하여 학습한 모델의 F1 점수를 Epoch 단위로 비교한 결과이다. 결과 그래프를 보면 알 수 있듯이 타입 정보를 추가한 데이터 형식을 사용하여 모델을 학습 했을 때 더 좋은 성능을 보인다는 것을 볼 수 있다. 첫째로 모든 데이터 셋에서 기존 연구보다 더 빨리 최적의 값으로 수렴한다. 그래프 결과와 같이 모든 Epoch의 F1 점수는 우리가 제안하는 데이터 형식에서 더 높은 결과를 보이며 이는 더 빨리 학습이 진행된다는 것을 유추할 수 있다.

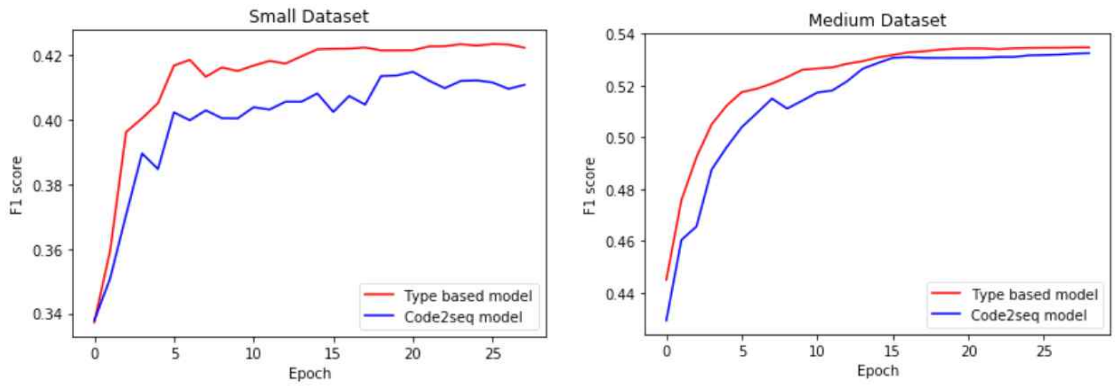


그림 20 Epoch에 따른 F1 점수 비교 그래프

그리고 둘째로 데이터 셋의 사이즈가 커질수록 차이는 미묘하지만 최적의 값이 타입 정보를 추가했을 때 더 향상된다. [표 3]은 각 데이터 셋을 30 Epoch까지 학습시켰을 때 가장 최적의 F1 점수를 나타낸다. 표와 같이 Small 데이터 셋을 사용했을 때 약 1%의 F1 점수가 더 높은 결과를 얻었으며 Medium 데이터 셋을 사용했을 때 0.2%의 더 높은 F1 점수를 얻을 수 있었다. 따라서 이를 통해 타입정보를 추가한 데이터 형식은 기존의 추상 구문 트리의 경로만을 이용한 데이터 셋보다 더 좋은 성능을 보인다는 것을 확인할 수 있다.

표 3 code2seq 모델과 타입 기반 모델의 성능 비교

	Code2seq model			Type based model		
	Precision	Recall	F1 Score	Precision	Recall	F1 Score
Small	45.70%	37.98%	41.48%	46.76%	38.69%	42.34%
Medium	59.76%	48.04%	53.26%	59.98%	48.23%	53.46%

나) 예제를 통한 기존 모델과의 비교

우리가 제안하는 타입 정보를 추가한 데이터 형식을 이용하여 코드 분석을 할 경우 코드 정보에 영향을 많이 받는 코드일수록 더 좋은 결과를 얻을 수 있다. [그림 21]은 타입 정보를 추가했을 때 더 좋은 결과를 얻는 예제를 보여준다. (a)는 Map의 특징이 코드에 영향을 받는 예제이다. Map은 (Key, Value)로 구성되어 있다. 즉, 리스트나 배열과 같이 순차적으로 데이터가 구성되어 있는 것이 아닌 Key를 통해 이에 부합되는 Value 값을 얻어낼 수 있는 형식으로 구성되어 있다. 따라서 Map에 따른 설정자와 접근자는 Value값과 관련되어야한다. (a)를 통해 두 모델을 비교해보면 기존의 Code2seq 모델은 Speaking 변수가 Map이라는 특징을 못 잡아 'setStudent'의 함수명을 예측한 것을 볼 수 있다. 반면에 타입 정보를 추가한 모델의 경우 Speaking 변수가 Map이라는 특징을 잡아 Map에 데이터를 추가하는 코드임을 분석하고 'setLevel'의 함수명을 예측하였다. 따라서 타입 정보를 추가함으로써 Map의 특징을 살려 코드를 분석하였다고 볼 수 있다. 비슷하게 (b)는 List의 특징이 코드에 영향을 받는 예제이다. merge 함수는 ArrayList에서 배열을 병합하는 함수이다. 즉, 리스트

A와 리스트 B의 데이터를 리스트 C에 입력하는 것이다. 따라서 함수의 이름은 merge가 적절하며 타입 정보를 추가한 데이터 형식의 경우 변수 A와 B가 List라는 정보가 강조되어 함수의 내용이 merge하는 역할을 한다는 것을 분석할 수 있으며 함수명을 ‘merge’라고 예측하였다. 하지만 기존의 Code2seq 모델의 경우 각 변수가 List라는 정보는 알고 있지만 강조가 되지 못하였기 때문에 함수의 내용이 병합하는 역할을 한다는 것을 분석하지 못하였다. 이와 같이 우리가 제안하는 타입 정보를 추가한 데이터 형식을 사용한 경우 타입 정보에 영향을 많이 받는 소스코드의 경우 코드의 특징을 추출하는데 더욱 유리하다는 것을 확인할 수 있었다.

```java
package Students;

import java.util.HashMap;

public class English {
    private HashMap<Student, String> Speaking = new HashMap<>();
    public void setLevel(Student student, String Level) {
        Speaking.put(student, Level);
    }
}
```

Code2seq model Prediction	Type base model Prediction
setStudent	setLevel

(a) Map 특징 강조

```java
import java.util.List;

public class ArrayList {
    public static void merge(List<Integer> A, List<Integer> B, List<Integer> C) {
        int pa = 0;
        int pb = 0;
        while (pa < A.size() && pb < B.size()) {
            if (A.get(pa) <= B.get(pb)) {
                C.add(A.get(pa++));
            } else {
                C.add(B.get(pb++));
            }
        }
        while (pa < A.size()) {
            C.add(A.get(pa++));
        }
        while (pb < B.size()) {
            C.add(B.get(pb++));
        }
    }
}
```

Code2seq model Prediction	Type base model Prediction
add	merge

(b) List 특징 강조

그림 21 타입 정보에 영향을 받은 코드 예제

나. 논의사항

1) 컴퓨팅 파워에 따른 성능 저하

우리는 기존 연구에서 사용하는 셋업과 다른 셋업을 사용하였다. 특히 우리는 Batch 크기를 기존 연구보다 현저하게 낮췄는데 이는 GPU 메모리의 부족으로 인한 한계에 따른 결정이었다. Batch 크기란 한 번의 Iteration동안 몇 개의 데이터를 묶어서 학습을 하는지에 대한 기준이다. 따라서 GPU의 메모리가 부족하다면 Batch 크기를 줄여야하며 이에 대한 영향으로는 학습 속도가 느려지며 오버슈팅 문제가 발생할 수 있다. [그림 22]의 (a)는 오버슈팅 문제를 보여준다. 오버슈팅이란 학습이 최적의 값으로 수렴해야 하는 것과 반대로 학습률이 너무 높아 무한으로 발산하는 것을 뜻한다. 따라서 Batch 크기의 감소로 인한 오버슈팅을 막기 위해 학습률을 낮춰 문제를 해결해야한다.

우리는 Batch 크기를 현저하게 줄인 만큼 오버슈팅 문제를 직면하였으며 이를 해결하기 위해 학습률 역시 현저하게 내려 학습을 진행하였다. 학습률을 내리면 오버슈팅 문제를 해결할 수 있지만 [그림 22]의 (b)와 같이 지역 극소(Local Minima) 문제가 발생한다. 지역 극소 문제란 그림과 같이 더욱 좋은 결과인 최적의 전역 극소(Global Minima) 값을 구하지 못하고 이보다 안 좋은 지역 극소를 찾아 수렴하는 문제를 뜻한다. 따라서 최적의 값보다 더 성능이 안 좋은 결과로 학습이 되며 우리는 이러한 결과를 얻었다. 특히 Large 데이터 셋에서 이 문제가 심각하게 드러났는데 기존 연구의 데이터 형식 및 모델을 이용하여 학습했을 때 약 59%의 F1 점수를 얻어야 하는데 반해 약 53%의 F1 점수로 수렴하는 결과를 얻었다.

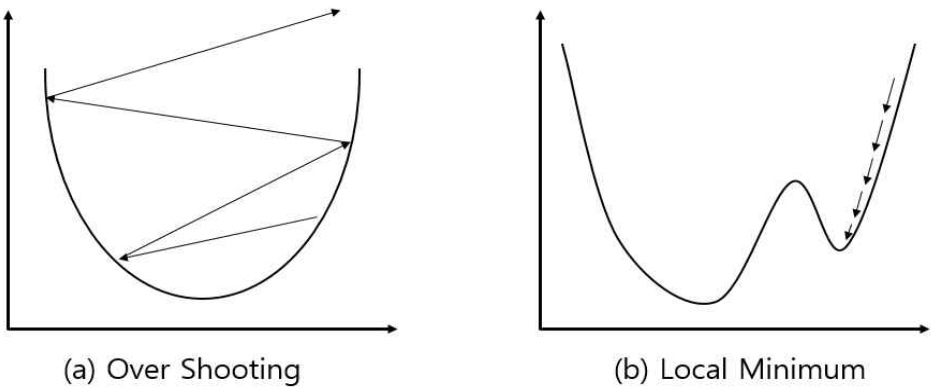


그림 22 오버슈팅과 지역 극소 문제

2) 타입 정보 추출의 확장

우리는 본 연구를 통하여 학습 데이터에 타입 정보를 추가하면 더욱 좋은 결과를 얻을 수 있다는 것을 알았으며 이는 타입 정보가 많으면 많을수록 결과는 더욱 좋아질 것이라는 예상을 할 수 있다. 하지만 본 연구에서는 필드 변수와 지역 변수에서 얻을 수 있는 정보만을 경로에 추가하여 사용하였지만 이 외에도 많은 곳에서 타입 정보를 얻을 수 있다. [그림 23]은 본 연구에서 추가한 타입 정보와 추가하지 못한 타입 정보를 보여준다. 초록 선은 타입 정보를 추출한 식별자를 나타내며 빨간 선은 타입 정보를 추출하지 못한 식별자를 나타내며 본 연구에서는 함수 리턴 타입 정보와 외부 클래스의 식별자 타입 정보를 추출하지 못하였다.

21줄과 22줄을 보면 모든 식별자가 필드변수와 함수의 파라미터를 이용하여 추출할 수 있다. 하지만 본 연구는 함수의 리턴 타입 정보는 추출하지 않았기 때문에 27줄의 extractSingleFile()와 같이 함수 리턴 타입 정보는 알 수 없다. 즉, [그림 23]의 코드를 본 연구에서 설계한 데이터 형식으로 전처리를 했을 경우 extractSingleFile()는 실제로 ArrayList<ProgramFeatures> 타입 정보를 갖고 있지만 경로의 extractSingleFile()에 해당하는 노드에는 이와 같은 타입 정보를 담고 있지 않다는 것이다. 따라서 향후 연구에서 필드 함수의 타입 정보를 함께 추출하여 함수를 호출하는 노드에 리턴 타입 정보 노드를 추가함으로써 함수 식별자의 정보를 고도화할 수 있다.

비슷하게 [그림 23]의 43줄은 추출하지 못한 외부 클래스의 식별자 타입 정보를 나타낸다. 3줄에서 CommandLineValue 클래스를 import했으며 해당 클래스의 MaxFileLength 변수를 사용하였다. 하지만 본 연구는 파싱하는 과정에서 클래스 내의 필드변수와 지역변수의 타입 정보만을 추출하여 사용하기 때문에 다른 클래스에 존재하는 식별자의 타입 정보를 알 수 없다. 함수의 리턴 타입 정보와 마찬가지로 MaxFileLength 변수는 int의 타입 정보를 갖고 있지만 데이터로 사용한 경로의 MaxFileLength 변수에 해당하는 노드에는 이와 같은 타입 정보를 갖고 있지 않는다. 따라서 향후의 다른 클래스의 식별자 정보를 사용할 때 타입 정보를 유추해서 경로에 추가한다면 더욱 더 데이터를 고도화 시킬 수 있을 것이며 이는 모델의 성능을 향상시킬 수 있을 것이다.

```java
import JavaExtractor.Common.CommandLineValues;

class ExtractFeaturesTask implements Callable<Void> {
    private final CommandLineValues m_CommandLineValues;
    private final Path filePath;

    public ExtractFeaturesTask(CommandLineValues commandLineValues, Path path) {
        m_CommandLineValues = commandLineValues;
        this.filePath = path;
    }
    public void processFile() {
        ArrayList<ProgramFeatures> features;
        try {
            features = extractSingleFile();
        } catch (IOException e) {
            e.printStackTrace();
            return;
        }
        if (features == null) {
            return;
        }
        String toPrint = featuresToString(features);
        if (toPrint.length() > 0) {
            System.out.println(toPrint);
        }
    }
    private ArrayList<ProgramFeatures> extractSingleFile() throws IOException {
        String code;

        if (m_CommandLineValues.MaxFileLength > 0 &&
                Files.lines(filePath, Charset.defaultCharset()).count() > m_Commanc
            return new ArrayList<>();
        }
```

그림 23 타입 추출 가능 및 불가능 식별자 구분

5. 결론 및 향후연구

본 논문에서는 추상 구문 트리에서 경로 정보에 타입 정보를 추가한 데이터를 이용하여 코드의 특징을 추출하여 분석하는 연구를 진행하였다. 소스코드에서 타입 정보는 코드에서 사용된 식별자들의 특징을 나타내며 식별자들의 특징 추출은 코드 전체 내용 특징에 직결되었다. 우리는 이 사실을 증명하기 위해 기존의 추상 구문 트리의 경로만을 이용하여 특징을 추출하는 모델과 타입 정보를 추가한 데이터를 이용하여 특징을 추출한 모델의 성능을 평가하였다. 데이터 셋은 기존 연구에서 사용한 Github의 Java 소스코드를 Small 크기와 Medium 크기로 구분하여 사용하였으며 각 데이터 셋을 이용하여 학습한 모델의 최적의 F1 점수를 이용하여 각 모델의 성능을 비교하였다. 결과를 통해 타입 정보를 추가했을 때 두 가지의 효과를 볼 수 있었다. 첫째로 기존 연구보다 더 빨리 최적의 값을 찾을 수 있었다. 그리고 둘째로 미미하지만 동일한 조건에서 더 좋은 최적의 값을 얻을 수 있었다. 이는 타입 정보에 영향을 많이 받는 소스코드 일수록 코드의 특징을 추출하는데 더 좋은 결과를 얻기 때문에 가능하였다. 이를 통해 소스코드을 분석하는데 타입 정보가 영향을 미친다는 것을 알 수 있다.

본 연구는 타입 정보를 소스코드에서 필드변수와 지역변수만을 이용하여 추출하였다. 하지만 타입 정보는 이 뿐만 아니라 필드 함수의 타입 정보와 호출된 클래스의 타입 정보가 있다. 우리는 향후 연구로 소스코드에서 추출할 수 있는 최대한의 타입 정보를 추가적으로 추출하여 데이터를 더 고도화할 필요가 있다. 또한 Large 데이터 셋에 대한 지역 극소 문제를 해결해야한다. 컴퓨팅 파워에 따른 성능 저하 문제는 컴퓨팅 환경을 상향시켜 해결하는 방법도 있지만 초기값과 같은 하이퍼 파라미터 값의 변경 또는 모델의 수정을 통해 해결할 수 있다. 우리는 다양한 시도를 통하여 최적의 하이퍼 파라미터 값을 찾아 지역 극소 문제를 해결하고 성능을 향상시킬 수 있는 모델에 대한 연구가 필요하다.

참고문헌

[1] D. Engler, D. Chen, S. Hallem et al., “Bugs as deviant behavior: A general approach to inferring errors in systems code,” ACM SIGOPS Operating Systems Review, vol. 35, no. 5, pp. 57-72, 2001

[2] M. Allamanis and C. A. Sutton, “Mining source code repositories at massive scale using language modeling,” in MSR, 2013, pp. 207-216.

[3] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating Sequences from Structured Representations of Code. In Proceedings of ICLR 2019.

[4] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Mapping language to code in programmatic context. In Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, pages 1643-1652. Association for Computational Linguistics. 2018.

[5] T. Kamiya, S. Kusumoto, and K. Inoue, “CCFinder: a multilinguistic token-based code clone detection system for large scale source code,” IEEE Transactions on Software Engineering, pp. 654-670, 2002.

[6] Hoa Khanh Dam, Truyen Tran, and Trang Pham. A deep language model for software code. In Workshop on Naturalness of Software (NL+SE), co-located with the 24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE). 2016.

[7] Mou, L., Li, G., Zhang, L., Wang, T., and Jin, Z.. Convolutional neural networks over tree structures for programming language processing. In Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, 2016.

[8] Allamanis, M., Brockschmidt, M., and Khademi, M. Learning to represent programs with graphs. In Proceedings of the International Conference on Learning Representations (ICLR). 2018.

[9] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec:

Learning distributed representations of code. Proc. ACM Program. Lang., 3(POPL):40:1-40:29, January 2019. ISSN 2475-1421.

[10] Vincent J Hellendoorn, Christian Bird, Earl T Barr, and Miltiadis Allamanis. Deep learning type inference. In Proceedings of the 2018 26th ACM Joint Meeting on European So□ware Engineering Conference and Symposium on the Foundations of So□ware Engineering. ACM, 152 -162. 2018.

[11] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. NL2Type: inferring JavaScript function types from natural language information. In Proceedings of the 41st International Conference on So□ware Engineering. IEEE Press, 304-315. 2019.

[12] M. Gabbrielli, S. Martini, Programming Languages: Principles and Paradigms, London, Springer-Verlag, 2010.

[13] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. A general path-based representation for predicting program properties. In Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, pages 404-419,

[14] Hochreiter, S. and Schmidhuber, J. Long short-term memory. Neural Computation, 9(8), 1735-1780. 1997.

[15] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. In ICLR. 2015.

[16] Javaparser [Online], https://javaparser.org/

[17] RicoSennrich, BarryHaddow, andAlexandra Birch. Neural machinetranslation of rarewords with subword units. In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 1715-1725, Berlin, Germany, August 2016. Association for Computational Linguistics

Deep learning-based understanding code that uses features that add type information to paths between terminal nodes in an abstract syntax tree

JinTaeck, Lim

Department of Electronics and Cmputer Engineering

Graduate School Chonnam National University

(Supervised by Professor Kwang Hoon Choi)

(Abstract)

Recently, with the activation of open source markets such as StackOverflow and Github, there are increasing cases of searching for various codes online and writing programs. Currently, however, code can only be searched with a limited set of search terms. That is, code that does not match the pattern suggested by the keyword cannot be searched. This difficulty has resulted in increasing the time required to utilize open source, and in order to solve this, a study is needed to extract and analyze the features of the code.

Meanwhile, deep learning technology that finds sophisticated patterns based on data has been spotlighted in the field of artificial intelligence . In particular, deep learning technology obtains more sophisticated results as the amount of data increases, which has the advantage of being useful in the open source market with large-scale source code. By learning the source code in a deep learning model, the characteristics of each source code can be extracted, thereby improving the environment of the open source.

In this paper, we study an improved code feature extraction method that analyzes source code using deep learning technology. Among the studies related to this, we proceed based on the code2seq study that extracts features using the path of the abstract syntax tree (AST). Since the abstract syntax tree is constructed through the parsing of the code, the relationship between each token of the code can be understood, which is advantageous for extracting features of the code.

We improve the performance of extracting features by advancing data by adding type information to code information obtained with an abstract syntax tree. In the source code, type information indicates the features of the identifier, which is directly linked to the features of the entire contents of the code. Therefore, the data with type information added to the path information of the abstract syntax tree allows the deep learning model to obtain better results in extracting the features of the code.

We improve the performance of extracting features by advancing data by adding type information to code information obtained with an abstract syntax tree. In the source code, type information indicates the characteristics of the identifier, which is directly linked to the characteristics of the entire contents of the code. Therefore, the data with type information added to the path information of the abstract syntax tree enables the deep learning model to obtain better results in extracting the features of the code.

This allows you to do two things. The first task is a function name prediction task. The function name prediction task is a task of predicting the function name through the function code, which can unify the function name rules and speed up development. When using the data set used in code2seq and comparing the performance with the existing studies, the data format with type information was able to obtain faster optimal value convergence and higher F1 score. The second task is the function summary task. Function summarization is the operation of summarizing sentence through the code of a function, and this can shorten the time of using open

source. As the data set, we used the CONCODE data set using JavaDoc-style annotations, and we showed that it is possible to summarize the function code by extracting the features of the function code in a sentence.

부록A. 구현 상세사항

부록A에서는 본 연구에서 사용한 컴퓨팅 환경과 환경 변수에 대해서 정리한다. 가. 에서 모델을 학습한 고성능 컴퓨팅 서버에 대해서 정리하며 나. 에서 학습할 때 사용한 환경 변수 옵션들에 대해서 정리한다.

가. 컴퓨팅 환경

본 연구는 고성능 컴퓨팅 서버 지원을 받아 진행되었다. 서버는 도커(Docker)를 이용하여 각 호스트마다 컨테이너를 할당해 주었으며 서버의 주요 스펙과 우리에게 할당된 컨테이너의 주요 스펙은 [표 4]와 같다.

표 4 서버 및 컨테이너 주요 스펙

서버 주요 스펙	- GPU : Tesla V100 32GB - HDD : 700GB - CPU : Xeon Gold 6126(2.6HGz) 12코어 - RAM : 96GB
컨테이너 주요 스펙	컨테이너OS : Ubuntu 18.04.2 LTS 파이썬 버전 : Python 3.6.8 파이썬 주요 패키지 : - tensorflow-gpu 1.14.0 - tensorflow-estimator 1.14.0 - tensorboard 1.14.0 - Keras-Applications 1.0.8 - Keras-Preprocessing 1.1.0 - matplotlib 3.1.1 - numpy 1.17.2 - rouge 0.3.2

나. 환경 설정

본 연구의 모델은 다양한 환경 변수를 사용하여 학습을 진행한다. [표 5]는 본 연구에서 사용한 모델의 환경 변수이다. 이 환경 변수는 [표 4]의 스펙에서 학습이 가능한 환경 변수들이며 스펙에 따라 일부 조정하여 사용할 수 있다.

표 5 모델의 환경 변수

config.py

config.BATCH_SIZE = 16
config.TEST_BATCH_SIZE = 16
config.READER_NUM_PARALLEL_BATCHES = 1
config.SHUFFLE_BUFFER_SIZE = 10000
config.CSV_BUFFER_SIZE = 100 * 1024 * 1024 # 100 MB
config.MAX_CONTEXTS = 1000
config.SUBTOKENS_VOCAB_MAX_SIZE = 190000
config.TARGET_VOCAB_MAX_SIZE = 27000
config.EMBEDDINGS_SIZE = 128
config.RNN_SIZE = 128 * 2
config.DECODER_SIZE = 320
config.NUM_DECODER_LAYERS = 1
config.MAX_PATH_LENGTH = 14 + 1
config.MAX_NAME_PARTS = 5
config.MAX_TARGET_PARTS = 6
config.EMBEDDINGS_DROPOUT_KEEP_PROB = 0.75
config.RNN_DROPOUT_KEEP_PROB = 0.5
config.BIRNN = True
config.RANDOM_CONTEXTS = True
config.BEAM_WIDTH = 0
config.USE_MOMENTUM = True
config.WORDS_MIN_COUNT = 20
 	
#Captioning Configuration
config.TARGET_WORDS_MIN_COUNT = 2
config.NUM_EXAMPLES = 52299
config.EMBEDDINGS_SIZE = 128 * 4
config.RNN_SIZE = 128 * 4
config.DECODER_SIZE = 512
config.MAX_TARGET_PARTS = 37
config.EMBEDDINGS_DROPOUT_KEEP_PROB = 0.3
config.RNN_DROPOUT_KEEP_PROB = 0.75

부록B. How to setup & run

부록B에서는 본 연구에서 제안한 코드 특징 추출 모델을 사용하는 방법에 대해서 소개한다. [그림 24]는 모델을 사용하는 과정을 도식화한 것이다. 가. 에서 데이터와 모델 준비 및 데이터 전처리 방법을 소개하고 나. 에서 이를 이용하여 모델을 학습하는 방법과 학습된 모델을 이용하여 예측하는 방법을 설명한다.

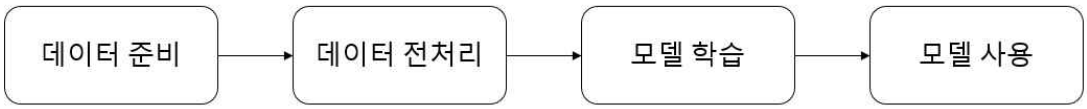


그림 24 모델 사용 과정

가. 데이터 및 모델 준비

본문에서 설명한 바와 같이 본 연구에서는 [3]의 데이터 셋과 [4]의 데이터 셋을 사용하였다. 각각의 데이터 셋은 해당 Github 주소[1)2)]를 통하여 얻을 수 있다. 또한 함수명 예측 작업의 경우 직접 데이터 셋을 구성하여 사용할 수 있다. 기존에 사용하는 데이터 셋은 Github의 API를 이용하여 직접 소스 코드 데이터 셋을 수집한 것이므로 이와 유사하게 Github에서 제공하는 API를 이용하면 다양한 언어 환경의 소스코드를 수집할 수 있다. 하지만 함수 요약 작업의 경우 이에 적합한 데이터 구성이 필요하다. 즉, 코드마다 코드를 요약한 문장이 함께 제공되는 데이터 셋이 필요하며 이를 위한 데이터 셋 구성에 대한 연구가 꾸준히 진행되고 있다. 우리는 다양한 데이터 셋 중 우리 연구에 가장 적합하다고 판단한 [4]의 데이터 셋을 사용하였다.

데이터가 준비가 되었다면 전처리기과 모델을 준비해야한다. 우리는 우리가 사용한 전처리기와 모델을 Github[3)]에 등록해놓았으며 자유롭게 사용할 수 있다. 전처리기의 경우 함수명 예측 작업과 함수 요약 작업의 전처리기를 각각 제공하고 있으며 원하는 작업에 따라 적합한 전처리기를 선택하여 사용하면 된다. 따라서 데이터 전

1) https://github.com/tech-srl/code2seq
2) https://github.com/sriniiyer/concode
3) https://github.com/limjintack/Type-based-code-analyze

처리를 진행하기 전에 interactive_predict.py에서 전처리기를 선택해야한다. 여기서는 편의를 위해 함수명 예측 작업의 Small 데이터 셋을 이용한 학습과정을 설명하며 따라서 전처리기는 JavaExtractor.jar를 사용해야한다. Small 데이터 셋과 전처리기가 준비 되어 있다면 아래의 명령을 통하여 데이터 전처리를 한다.

```
bash preprocess.sh
```

전처리가 끝나면 학습 데이터로 사용될 임의의 확장자인 c2s 파일들이 생성될 것이다. 간혹 데이터 셋의 폴더 경로가 일치하지 않아 에러가 날 수 있으니 이에 유의하여 전처리 과정을 진행해야 한다.

나. 모델 학습 및 예측 작업

전처리가 끝난 데이터가 준비되어 있다면 이를 이용하여 딥러닝 모델을 학습하여야한다. 학습을 진행하기 전 컴퓨팅 스펙에 따른 환경 변수를 조정해야한다. 즉, 환경에 맞춰 [표 5]에서 나열한 환경 변수와 모델의 Learning Rate를 수정해야한다. 여기서 Learning Rate는 model.py의 “tf.train.exponential_decay” 함수에서 조정하여 사용되며 본문 4의 나. 에서 설명한 오버슈팅을 막는 적절한 값이 필요하다. 우리가 사용한 연구의 경우 보통 0.004의 학습률을 사용하였으며 환경 변수의 Batch Size는 16 또는 32를 사용하였다. 컴퓨팅 스펙에 따라 환경 변수를 조정하였다면 앞서 설명한 전처리가 끝난 5개의 c2s 파일을 이용하여 모델을 학습하기 위해 아래와 같은 명령을 이용한다.

```
bash train.sh
```

명령을 입력하면 학습이 진행될 것이며 [표 4] 컴퓨팅 환경에서 Small 데이터 셋의 경우 1 Epoch당 약 1시간이 소요됐으며 [그림 20]과 같이 15 Epoch에서 수렴하는 것을 확인할 수 있었다.

모델 학습이 완료가 되면 이를 이용하여 함수명 예측 작업을 수행할 수 있다. 해당 디렉토리에 있는 Input.java 파일에 예측하고자 하는 함수의 코드를 작성하고 아래의 명령어를 통해 예측 작업을 진행한다. 모델은 총 20 Epoch 학습이 되었다는 가정하에 20번째 학습된 모델을 사용하기 때문에 model_iter20 파일을 불러 들어와 사용한다.

```
python code2seq.py --load models/java_small/model_iter20 --predict
```

명령을 입력하면 [그림 25]와 같이 적절한 코드의 함수명이 예측되는 것을 볼 수

있다.

추가적으로 [표 5]의 환경 변수 중 BEAM_WIDTH를 수정하여 후보 함수명을 출력할 수 있다. 즉, BEAM_WIDTH를 0에서 5로 수정하면 [그림 26]과 같이 가장 유력하다고 예측한 merge를 포함하여 다음으로 유력하다고 예측한 후보 함수명들을 출력할 수 있다.

```
Original name:  merge
Predicted:      ['merge']
Attention:
TIMESTEP: 0     : merge
0.021588        context: b,VDID0|Prm|List<Integer>|Mth|Prm|List<Integer>|Cls|Prim0,integer
0.020922        context: b,VDID0|Prm|List<Integer>|Mth|Prm|List<Integer>|VDID0,c
0.019286        context: void,Void0|void|Mth|Nm1,METHOD_NAME
0.016905        context: a,VDID0|Prm|List<Integer>|Mth|Prm|List<Integer>|VDID0,b
0.015087        context: a,VDID0|Prm|List<Integer>|Mth|Prm|List<Integer>|VDID0,c
0.014788        context: a,VDID0|Prm|List<Integer>|Mth|Prm|List<Integer>|Cls|Prim0,integer
0.013715        context: void,Void0|void|Mth|Prm|List<Integer>|Cls|Prim0,integer
0.012193        context: b,VDID0|Prm|List<Integer>|Cls|Prim0,integer
0.009854        context: c,VDID0|Prm|List<Integer>|Cls|Prim0,integer
0.009240        context: a,VDID0|Prm|List<Integer>|Cls|Prim0,integer
```

그림 25 모델 사용 결과

```
Original name:  merge
Predicted:
        ['merge']
        ['add']
        ['join']
        ['compare']
        ['map']
```

그림 26 예측한 예비 후보 결과

감사의 글

2020년 8월

임 진 택