2006 한국컴퓨터종합학술대회 논문집 Vol. 33, No. 1(B)

Parse Tree Kernel을 이용한 소스코드 표절 검출

손정우⁰ 박성배 이상조 박세영
경북대학교 컴퓨터공학과
{jwson⁰, sbpark, sjlee}@sejong.knu.ac.kr, seyoung@knu.ac.kr

Program Plagiarism Detection Using Parse Tree Kernels

Jeong-Woo Son⁰, Seong-Bae Park, Sang-Jo Lee, Seyoung Park
Dept. of Computer Engineering, Kyungpook National University

요 약

표절이란 원작자의 허락 없이 저작물의 일부분 혹은 전체를 사용하는 것이다. 이는 특히 대학의 프로그래밍 코스에서 심각한 문제가 된다. 이를 해결하기 위해 많은 표절 검출 시스템이 연구되어 왔으나 복사된 소스코드에 필요 없는 코드를 첨가할 경우, 성능이 낮아지는 문제가 있었다. 이 문제는 기존 시스템이 소스코드의 구조적인 정보를 효율적으로 다루지 않았기 때문이다. 본 논문에서는 Parse Tree Kernels를 이용한 소스 코드 표절 검출 시스템을 제안한다. 제안한 시스템은 Parse Tree Kernels를 이용하여 소스 코드의 구조적 정보를 효과적으로 다룬다. 이를 보이기 위한 실험에서는 기존의 표절 검출 시스템인 SID, JPlag와 비교하여 제안한 시스템이 소스 코드의 구조적 정보를 기존 시스템에 비해 효율적으로 이용하고 있음을 보였다.

1. 서 론

표절은 특히 교육 환경에서 심각한 문제로서 표절물을 검출하는 것은 교육자들이 반드시 해야 할 일 중 하나이다. 하지만 일일이 표절물을 검출하는 것은 많은 시간과 노력을 필요로 하는 일이다. 이로 인해 표절물을 자동으로 검출하는 시스템은 1970년대 중반부터 꾸준히 연구 되어 왔다.

프로그램 표절 검출 시스템은 크게 두가지로 나뉜다. 첫 번째는 Attribute counting metric 시스템으로, 오퍼레이터와 오퍼랜드, 고유의 오퍼레이터와 고유의 오퍼랜드로 이루어진 벡터를 이용하여 프로그램을 표현하고 비교한다. Attribute counting metric 시스템으로는 Halstead가 제안한 software science metric을 이용한 표절 검출 시스템[1]이 있으며, Berghel[2], 그리고 Grier[3]가 제안한 시스템이 있다. 이러한 시스템은 구조적 정보를 이용한 최근의 시스템과 비교해서 낮은 성능을 보인다.

두 번째 표절 검출 시스템은 Attribute counting metric과 소스 코드의 구조를 모두 이용한다. 이 시스템에서는 소스 코드의 구조적 정보를 string으로 표현을 한다[4,5]. Prechelt는 greedy string tiling 알고리즘을 이용한 JPlag 시스템[6]을 제안하였으며, Chen은 Kolmogorov complexity를 이용한 SID 시스템[7]을 제안하였다. 이들 시스템은 소스 코드의 구조적 정보를 부분적으로 이용하여 좋은 성능을 보였지만, 시스템에서 구조적 정보를 모두를 반영하지 않았다. 그 결과 소스 코드에 필요 없는 코드를 삽입하는 기법에 취약하다.

소스 코드의 구조적 정보를 효율적으로 반영하기 위해서 표절 검출 시스템은 코드의 구조적 정보를 가지는 parse tree를 이용하여야 한다. Parse tree를 이용하기 위해서는 feature space상에 정보의 손실 없이 사상하는 것이 중요하다. 하지만 tree와 같은 자료 구조는 사상하기 어렵다. 이를 해결하는 방법 중 하나가 Kernel methods이다. 이 중, Collins와 Duffy가 제안한 Parse tree kernel[8]은 parse tree를 다루는데 적합하다.

본 논문에서는 Parse tree kernel을 이용하여 자바언어로 만들어진 소스 코드의 표절 여부를 검출하는 시스템을 제안한다. 먼저 자바 소스 코드에서 parse tree를 추출한 후, 소스 코드 표절에 적합하도록 만들어진 Parse tree kernel을 이용하여 코드간의 유사도를 계산한다. 그 후, 표절일 가능성이 큰 소스 코드들을 제시하게 된다.

제안한 시스템을 평가하기 위해 하나의 소스 코드에 대해 크게 2가지 종류의 기법을 이용한 표절물들을 만들어 실험하였다.

본 논문의 구성은 다음과 같다. 2장에서는 제안한 시스템을 보여주고, 3장에서는 실험결과를 설명한다. 마지막으로 4장에서 결론을 기술한다.

2. 표절 검출 시스템

본 논문에서 제안하는 시스템은 3단계로 나누어 진다. 첫 번째 단계에서는 ANTLR[9]을 이용하여 소스 코드로부터 parse tree를 추출한다. 추출된 parse tree를 parse tree에 적용시켜 유사도를 얻게 된다. 마지막으로 유사도 값들을 정규화시켜 임계값을 넘어서는 코드에 대해 표절물로써 제시하게 된다. 그림 1은 본 논문에서 제안한 시스템의 구조를 보여준다.

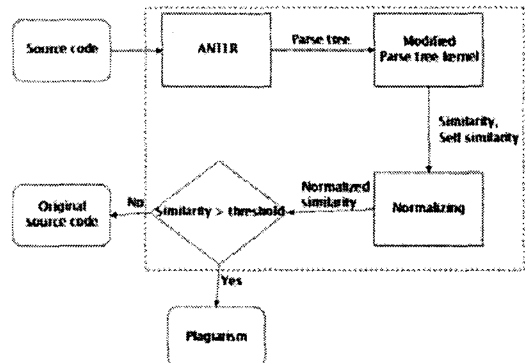

그림 1. 제안한 시스템의 구조.

2.1. Parse tree 추출

표절 검출에서 구조적 정보를 이용하기 위해서는 소스 코드로부터 parse tree를 추출해야 한다. 제안한 시스템에서는 이를 위해 ANTLR을 이용하고 있다. 특히, ANTLR의 tree parser는 소스 코드를 parse tree로 변환 하고, 출력한다.

2.2. Parse Tree Kernel

Parse tree kernel은 Convolution kernel[10]의 하나로 parse tree들을 다루는데 특화된 kernel이다. Parse tree kernel에서 vector의 feature는 각 parse tree에 나타날 수 있는 모든 subtree로 이루어진다. 이때, 각 feature의 값은 subtree의 빈도로 할당된다. 그림3은 간단한 parse tree에 대해 subtree들의 예를 보여 준다. 하지만 이러한 subtree를 명시적으로 구한다는 것은 불가능하다. 이에 Collins와 Duffy는 명시적인 열거 없이 내적을 구하는 방법을 제시하였다.

$Subtree_1$, $subtree_2$, ..., $subtree_n$을 parse tree T의 subtree라 하면 parse tree T는 다음과 같이 vector로 나타낼 수 있다.

$$V_T = (\#subtree_1(T), \#subtree_2(T), \ldots, \#subtree_n(T))$$

위 수식에서 $\#subtree_i(T)$는 $subtree_i$의 빈도수를 나타낸다.

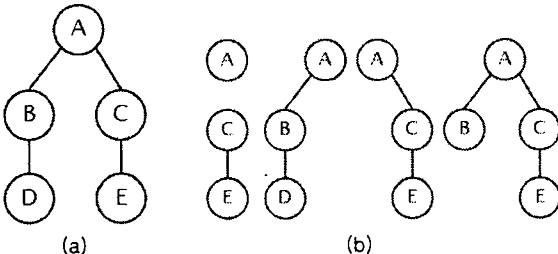

그림 3. (a) Parse tree (b) Parse tree (a)에 나타날 수 있는 subtree들.

두 parse tree T_1과 T_2사이의 내적은 아래와 같은 식으로 계산되어 진다.

$$\langle V_{T_1}, V_{T_2} \rangle = \sum_i \#subtree_i(T_1) \cdot \#subtree_i(T_2)$$
$$= \sum_i \Big(\sum_{n_1 \in N_{T_1}} I_{subtree_i}(n_1)\Big) \cdot \Big(\sum_{n_2 \in N_{T_2}} I_{subtree_i}(n_2)\Big)$$
$$= \sum_{n_1 \in N_{T_1}} \sum_{n_2 \in N_{T_2}} C(n_1, n_2)$$

위 식에서 N_{T_1}과 N_{T_2}는 T_1과 T_2의 모든 노드이며 indicator 함수 $I_{subtree_i}(n_1)$는 노드 n_1이 최상위 노드인 subtree가 있으면 1, 아니면 0을 반환한다. $C(n_1, n_2)$ 함수는 다음과 같이 정의 가능하다.

$$C(n_1, n_2) = \sum_i I_{f_i}(n_1) \cdot I_{f_i}(n_2)$$

함수 $C(n_1, n_2)$는 다음과 같은 재귀적인 규칙을 이용하여 polynotima time에 계산 가능하다.

규칙1. n_1과 n_2의 자식 노드들이 다르다면 $C(n_1, n_2) = 0$

규칙2. n_1과 n_2가 말단 노드라면 $C(n_1, n_2) = 1$

규칙3. 그 외

$$C(n_1, n_2) = \prod_i^{nc(n_1)} (1 + C(ch(n_1, i), ch(n_2, i))) \quad (1)$$

함수 $nc(n_1)$는 n_1의 자식노드의 수를 반환한다.

2.3. Modified Parse Tree Kernel

Parse tree kernel을 이용하여 소스 코드의 구조적 정보를 효율적으로 다룰 수는 있으나 표절 여부를 검출하는데 이용하기에는 몇가지 문제가 있다. 첫 번째 문제는 유사도의 값이 소스 코드의 크기가 늘어남에 따라 기하급수적으로 커진다는 것이다. 이를 해결하기 위해 본 논문에서는 재귀 알고리즘의 수식1을 로그를 취하여 다음과 같이 바꾸어 값이 기하급수로 커지는 것을 막았다.

$$C(n_1, n_2) = \log^L \prod_i^{nc(n_1)} (1 + C(ch(n_1, i), ch(n_2, i)))$$

두 번째 문제는 일반적인 Convolution kernel이 가지는 문제점으로 Parse tree kernel이 parse tree에 나타나는 모든 subtree를 쓰므로 인해서 조금이라도 다른 parse tree들에 대해 너무 낮은 값을 보여 준다는 것이다.

이 문제를 해결하기 위해 본 논문에서는 Parse tree kernel이 모든 subtree를 feature로 쓰지 않게 subtree의 depth를 제한했다. 이로 인해 소스 코드사이의 유사도와 자기 자신과 비교 했을 때의 유사도 차를 줄일 수 있다. 이를 구현하기 위해 Parse tree의 재귀 알고리즘의 규칙2를 다음과 같이 바꾼다.

규칙2. n_1과 n_2가 말단 노드이거나 현재 depth가 제한한 depth와 같다면, $C(n_1, n_2) = 1$

2.4. 표절 검출

제안한 시스템에서는 코드간의 유사도가 임계값을 넘을 경우 표절로 간주한다. 하지만 Parse tree kernel을 통해 얻어지는 값은 상한값이 없다. 이 때문에 유사도들을 정규화해줄 필요가 있다. S1과 S2를 소스 코드, K(S1, S2)를 Parse tree kernel을 이용하여 유사도를 반환하는 함수라 하면, 정규화 함수 N(S1, S2)는 다음과 같이 정의 가능하다.

$$N(S_1, S_2) = \frac{K(S_1, S_2)}{E(S_1, S_2)} \times 100$$

E(S1, S2)는 정규화 인수로써 다음과 같이 정의 된다.

$$E(S_1, S_2) = \frac{K(S_1, S_1) + K(S_2, S_2)}{2}$$

3. 실험

3.1 Data set

실험을 위한 데이터는 4명의 프로그래머에 의해 만들어 졌다. 각 프로그래머는 하나의 원본 소스 코드를 만든 후, 다음과 같은 표절 기법에 의거하여 변형하게 된다. 이때 각 기법

당 하나의 표절 소스코드가 만들어 진다.

1. 간단한 표절 기법
   (a) 변수, 함수, 클래스의 이름을 고친다.
   (b) 'for'나 'while'같은 함수를 같은 동작을 하는 다른 명령으로 바꾼다.
   (c) 함수나 클래스를 합치거나 나눈다.

2. 구조적 표절 기법
   (a) 80 라인의 필요 없는 코드를 삽입한다.
   (b) 120 라인의 필요 없는 코드를 삽입한다.
   (c) 660 라인의 필요 없는 코드를 삽입한다.

3.2 실험 결과

실험에서 subtree의 depth는 5로 제한했으며, 유사도의 임계치는 40으로 정했다. 표1은 간단한 표절 기법에 대한 제안한 시스템의 실험 결과를 보여준다. 4명의 프로그래머가 각 4개씩의 코드를 만들었기에 16개의 소스 코드가 있으며 표에서 굵게 표시된 숫자는 시스템이 표절로 판단한 코드들이다. 표1에서 보여 주듯이 제안한 시스템은 각 원본 소스코드에 대해 변형된 코드들을 모두 검출하였다.

Source Code	Group1	Group2	Group3	Grpup4
1	27	**100**	32	34
2	29	**55**	33	35
3	27	**45**	29	32
4	28	**45**	31	33
5	**100**	27	25	30
6	**47**	27	25	30
7	**60**	25	23	25
8	**51**	25	23	25
9	25	32	**100**	31
10	26	33	**47**	32
11	23	29	**70**	30
12	25	31	**58**	30
13	30	34	31	**100**
14	31	34	29	**49**
15	25	31	28	**62**
16	29	33	30	**64**

표 1. 간단한 표절기법에 대한 실험 결과

표2는 구조적 표절 기법에 대한 실험 결과를 보여준다. 실험 결과, 더 많은 코드들이 삽입될수록 유사도는 떨어지나, 제안한 시스템에서 계산한 유사도가 항상 임계치를 넘는 것을 알 수 있다. JPlag의 경우, 많은 라인의 코드가 삽입될 경우 표절을 검출하는데 실패하는 것으로 나타났다. SID의 경우 표절 검출에는 문제가 없으나 유사도가 너무 낮아 표절로써 결정을 내리는데 어려움이 있을 것으로 보인다. 결과적으로 제안한 시스템은 더 많은 코드들이 삽입될수록 더 좋은 성능을 보였다.

Number of added lines	Proposed Method	SID	JPlag
80	65.9	38	73.30
160	58.1	16	67.80
660	42.5	34	15.40

표 2. 구조적 표절기법에 대한 실험 결과

4. 결론

본 논문에서는 Parse tree kernel을 이용하여 소스 코드의 구조적 정보를 효율적으로 이용한 새로운 표절 검출 시스템을 제안하였다. 제안한 시스템에서는 ANTLR을 이용해 소스 코드의 parse tree를 추출하고 이들 사이의 유사도를 표절 검출에 맞게 고쳐진 Parse tree kernel을 이용하여 계산한다.

제안한 시스템을 검증하기위해 크게 두 가지 종류의 표절 기법에 대해 실험하였다. 간단한 표절기법에 대해 100%의 정확도를 보였으며, 비교 대상인 SID나 JPlag가 구조적 표절기법에 대해 취약한데 반해 제안한 시스템은 크게 영향을 받지 않는 것으로 나타났다.

본 논문에서 제안한 시스템은 프로그래밍 언어에 독립적이다. ANTLR을 이용하여 parse tree를 추출할 때 C나 C++, Pascal과 같은 언어에 대해 각 언어에 맞는 parser를 이용함으로써 표절을 검출할 수 있다.

참 고 문 헌

[1] M. Halstead, "Elements of Software Science", *Elsevier*, 1977.
[2] H. Berghel and D. Sallach, "Measurements of Program Similarity in Identical Task Environments", *SIGPLAN Notices*, Vol. 19, No. 8, pp. 65--75, 1984.
[3] S. Grier, "A Tool that Detects Plagiarism in Pascal Programs", *Twelfth SIGCSE Technical Symposium*, Vol. 13, No. 1, pp. 15--20, 1981.
[4] S. Schleimer, D. Wilkerson, and A. Aiken, "Winnowing: Local Algorithms for Document Fingerprinting", *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 76--85, 2003.
[5] M. Wise, "Detection of similarities in student programs: YAP'ing may be preferable to Plague'ing", *ACM SIGSCE Bulletin*, Vol. 24, No. 1, pp. 268--271, 1992.
[6] L. Prechelt, G. Malphol, and M. Philippsen, "Finding Plagiarisms among a Set of Programs with JPlag", *Journal of Universal Computer Science*, Vol. 8, No. 11, pp. 1016--1038, 2002.
[7] X. Chen, B. Francia, M. Li, B. McKinnon, and A. Seker, "Shared Information and Program Plagiarism Detection", *IEEE Transactions on Information Theory*, Vol. 50, No. 7, pp. 1545--1551, 2004.
[8] M. Collins and N. Duffy, "Convolution Kernels for Natural Language", *Proceedings of the 14th Neural Information Processing Systems*, 2001.
[9] T. Parr and R. Quong, "ANTLR: A Predicated-LL(k) Parser Generator", *Journal of Software Practice & Experience*, Vol. 25, No. 7, 1995.
[10] D. Haussler, "Convolution kernels on discrete structures", *Technical Report UCSC-CRL-99-19*, 1999.