<https://pakalguksu.github.io/2020/02/28/CS224n-5%EA%B0%95-Dependency-Parsing/>

<https://gnoej671.tistory.com/5>

<https://jeongukjae.github.io/posts/cs224n-lecture-5-dependency-parsing/>

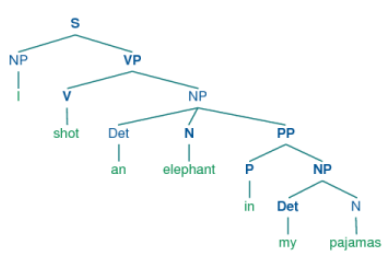
<https://lee6boy.wordpress.com/2013/06/28/parsing-dependency-parsing-graph-based-parsing%EC%9D%B4-%EB%AD%94%EA%B0%80/>

간단한 단어들이 base가 돼 문장을 이룬다.=> 따라서 여러 형태의 문장이 나올 수 있고, 문장의 구조를 분석하는 것은 문장을 정확히 이해하는데 필수적이다. = 문장이 무엇을 의미하는지 알 수 있다. = 문장 구조를 파악하는 이유

Pasring: 구문 분석 = 구조 분석 = 문장을 쪼개고 나누고 분석하고

문장 구조 파악 방법 2가지: Phrase structure와 dependency structure

Phrase structrue는 nested constituents(중첩된 구성요소)로 분류 / Context-free grammar(맥락 신경 안씀)와 비슷 = phrase를 이루고, phrase가 합쳐서 더 큰 phrase를 산출할 수 있다.  
=> 문장의 words의 구조를 품사등으로(카테고리) 파악한다.(part of speech, POS) / 명사구, 동사구 이런 것처럼 구단위로 묶어 구조를 파악



하지만 요새 대세는 아래 (언어에 상관없이 적용(CFG는 영어 문법에 특화) / 문장의 의미 구조 파악 가능(context based) / 다른 단어 와의 관계를 설명하려고 한다. = 이전에 word2vec의 distributed word representation과 비슷한 맥락)

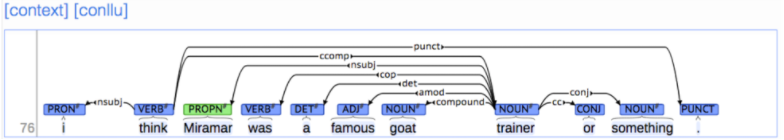
Dependency structure는 각 단어간의 의존 관계(dependency) => 문장 구조 파악  
ex) 동사의 목적어, 명사를 수식하는 절  
“Look in the large crate in the kitchen by the door”에서는 Look이 전체 문장의 root로 main 목적어인 crate는 Look에 의존적이다. = 이렇게 의존적인 형태를 이어나가 tree 형태로 구성할 수 있다.(Look은 superior & head / crate는 dependent & modifier)

품사의 종류와 절의 종류를 정의로 문장 구조 파악 = phrase structure / 단어간의 관계로 문장 구조 파악 = dependency structure

Det = Determiner / N = Noun / Adj = Adjective / NP = Noun Phrase / P = Preposition / PP = Prepositional Phrase

“San Jose cops kill man with knife” => (kill->cops, subj), (kill->man, obj), (kill->knife, nmod), (knife->with, case) // 하지만 중의적으로 (kill->knife, nmod)가 (man->knife, nmod)가 될수도 있다. = 문장의 의미가 모호(구조 분석에 따라, 무엇이 무엇을 modify 하냐에 따라) = 인간의 언어가 프로그래밍 언어랑 다른점(프로그래밍 언어는 hard rule로 애매한 부분이 없다.)

따라서 이런 dependency를 학습할 수 있도록 만든 데이터셋(=tree bank)라는 것이 있고, 이를 통해 학습한 모델은 parser, POS tagger 역할을 할 수 있다. (근데 아마 품사 정보도 있는 듯)



* Binary asymmetric arrow
* Arrow는 typed 이며 문법적인 관계
* 보통 fake ROOT를 추가
* 또한 reusable 하다.

Dependency structure => 기원전 5세기 / Constituency & CFG = 20세기

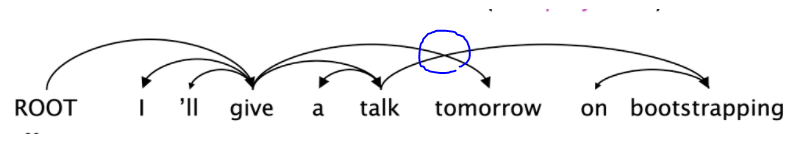
Dependency parsing = 문장에서 dependency structure를 찾은 과정( arrow 하나가 dependency 하나라고 보면 된다.)

* Bilexical affinities = 두 단어 사이의 실제 의미에 드러나는 관계를 의미  
  [discussions -> issues]와 같이 단어의 의미 상으로 dependency가 있을 법한 관계에 주목
* Dependency distance = dependency는 가까운 위치에서 나타난다 라는 사실에 주목
* Intervening material = dependent한 단어들 사이에 세미콜론 등의 관계를 나타내는 문자가 위치한다는 것을 고려(세미콜론이나 구둣점 뛰어 넘지 않음)
* Valency of Head = 어느정도는 기본 rule에 따름, 특정 단어 앞뒤로 가질 수 있는 dependency의 개수를 고려 = “was completed” 의 뒤에는 목적어가 올 수 없다, 앞에는 주어가 있을 것이다 등의 정보는 알 수 있다.(그런 상황에서 dependency를 파악)

의 특성을 고려하면서 parsing을 한다. 또한 문장의 맨앞에 ROOT라는 가상의 단어가 있고 “모든 word는 ROOT 또는 다른 단어에 dependent”라는 전제하에 parsing을 한다.

* ROOT에 dependent 되는 단어는 단 1개(동사 같은 것)
* A->B, B->A의 cycle dependent는 허용하지 않는다.

이 방식으로 dependencies를 tree로 나타낼 수 있다.

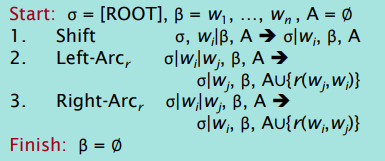


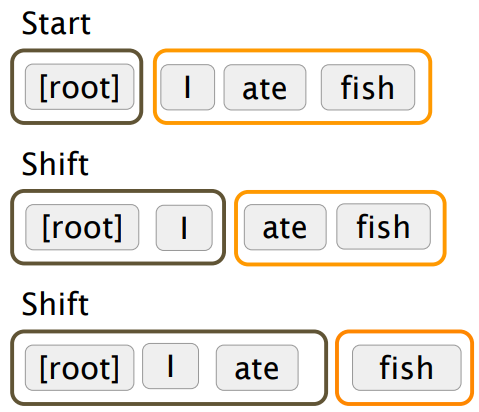
이렇게 cross가 발생한다 = non-projective = 보통 문장 parsing 할 때 허용할 수 있다.! (nesting constraint 라고 했음 = cross 주로 없다.) 하지만 위처럼 예외 허용할 수 있다.  
CFG 규칙으로 생성되는 dependency라면 projective하다.

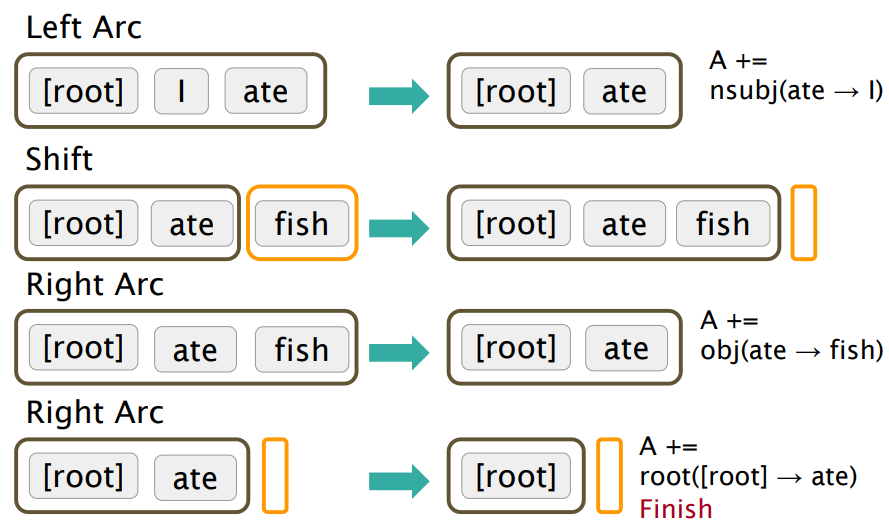
Dependency Parsing의 특징과 상황은 이전과 같고 문제를 해결하는 알고리즘이 몇 개 있다.(동적계획법, 그래프 알고리즘, 제약조건 만족)  
하지만 여기서는 “transition-based parsing”(변화, 이동 기반 파싱) or “deterministic dependency parsing”을 소개

일단, Greedy transition-based parsing을 알아보자. = 각 상황에서 최선의 transition을 반복해 문장을 parser = 이런 rule의 parser = Arc-standard transition-based parser(2003)

* stack에는 [root], 나머지는 buffer에 담는다.
* Action 3개  
  1. Shift: Buffer의 top word를 stack의 top position에 둔다.  
  2. Left arc: Stack의 top word를 화살표로 연결 후 dependent 제거  
  3. Right arc: stack의 top word를 화살표로 연결 후 dependent 제거
* 끝날 때는 stack에 단어 한 개[root], buffer은 empty 여야 한다.
* Automatic parser는 각 step 마다 1~3중 한 개를 선택하는 것이다.







하지만 만약 parser가 적절한 행동을 못한다면 dependency parsing 작업이 엉망일 것이다.

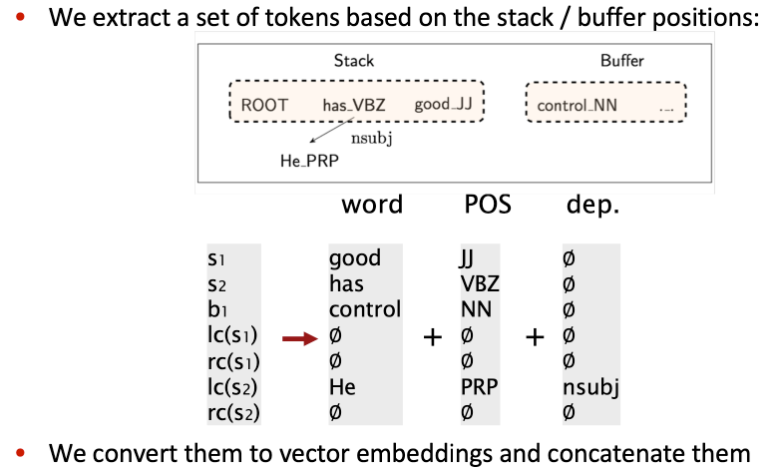
따라서 MaltParser는 action을 선택하는 classify를 위해 머신러닝 기술을 도입했다. => 가장 가능성 높은 행동 취함.(2005)

Feature는 다음과 같다. [top of stack word, 그 단어의 POS, top of buffer word, 그 단어의 POS ..] / 비록 SOTA는 아니지만 linear time으로 매우 빠르며 성능도 봐줄만 하다.

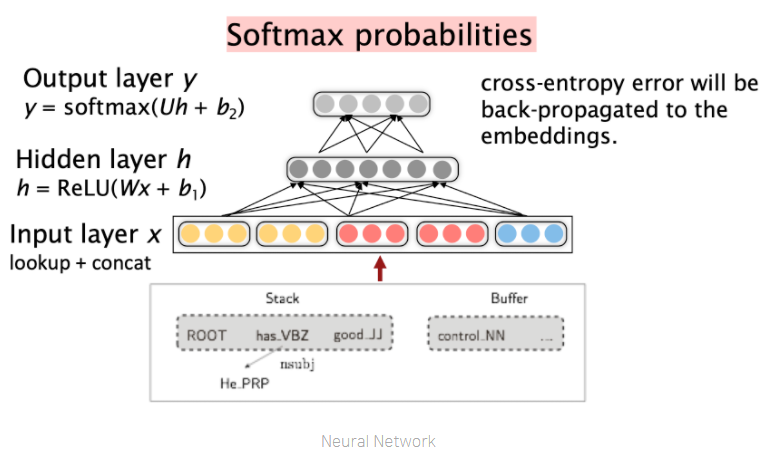
하지만 여기서 문제가 단어는 BoW로 다른 features는 one-hot vector로 나타낸다. => 그리고 concatenate 까지하면 매우 큰 입력 featrue가 되고 더군다나 sparse 하다. => neural net model 사용. (계산도 오래 걸림) => feature를 만드는데 95% 이상의 시간이 쓰임.

또한, 보지 못한 단어는 vector화 할 수 없기 때문에 feature로 줄 수가 없다. (embedding은 그냥 학습 데이터에 없었을지라도 걍 변환 가능) = incomplete

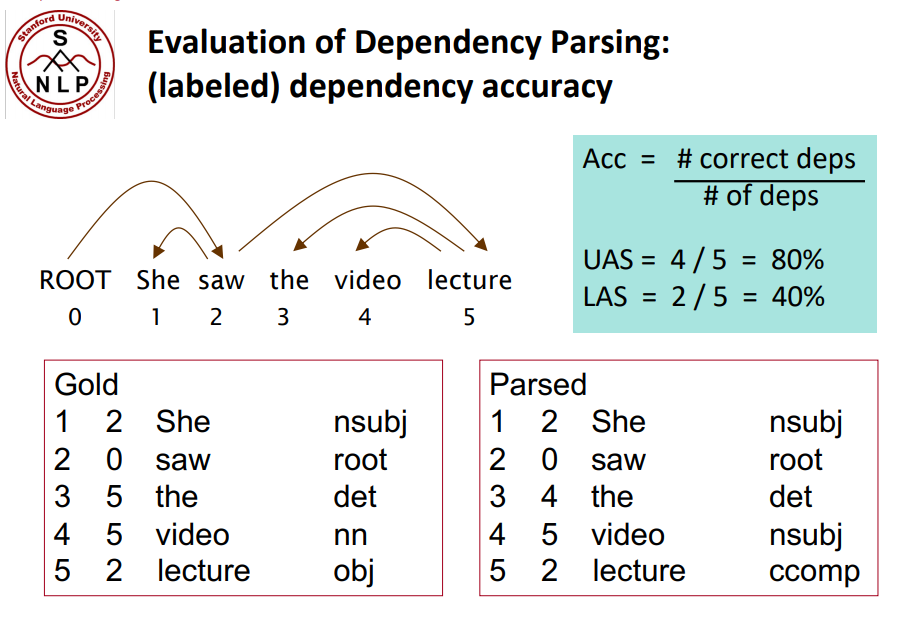
따라서 각 단어를 word embedding을 통해 d차원의 dense vector로 나타냄  
또한, POS tag와 dependency label도 d차원의 vector로 나타냄. (dependency label은 이전까지 모델이 예측 했던 arc decision)  
이것들을 concatenate 하여 input으로 준다.



위의 상황 하나하나가 데이터셋이 된다. 그리고 그때 알맞은 action은 label로 정해져 있다.



ReLU > tanh, sigmoid : vanishing gradient 때문에 / + exp 연산 / sigmoid는 심지어 slow convergence(not zero centered)



UAS(Unlabeled Attachment Score): dependency 관계가 제대로 됐는지만 고려  
LAS(Labeled Attachment Score): dependency label까지 제대로 됐는지 고려