Subramanian Shankar

CS410 Tech Review- Tree Kernels tools for NLP

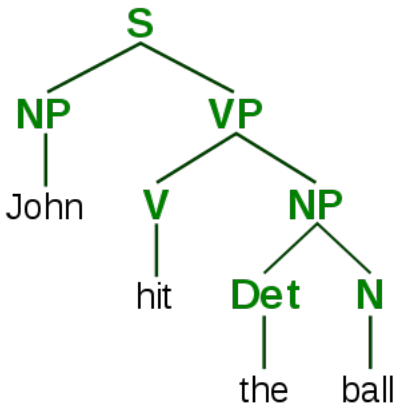
(sss117)

Background-

As a part of my tech review, I am doing analysis and review of existing tools and generate tree kernels for parse trees generated for syntactic parsing of documents. This tech review will initially introduce the basics of parsers and focus on how we can use them for software code. Parsers provide in-depth information of the data structures being used in the code and are being currently being explored in the current upcoming release. As a part of this, the techniques and tools being reviewed here will help in program analysis and program transformation systems , as well as detecting code plagiarism and code duplication.

Introduction-

To see what has changed in a document between two versions, it is necessary to compare parser trees for the two versions to check for their differences. A parse tree is an ordered, rooted tree that represents the syntactic structure of a string according to a context-free grammar. It is made up of different nodes and branches. A node is either a root node, a branch node, or a leaf node.



Lets say we have a sentence 'Jon hit the ball'. The figure on the left shows the parse tree representation of the sentencer, starting from S and ending in each of the leaf nodes (John,ball,the,hit). Here, S is a root node, NP and VP are branch nodes, while John, ball, the, and hit are all leaf nodes.

As we can see, these trees can grow to be very large and incredibly complex, as the document length increases, and utilization of simple techniques of dot products becomes impractical over lengthy trees. The accuracy provided by using trees can also become lower than the one provided by linear models on manually designed features.

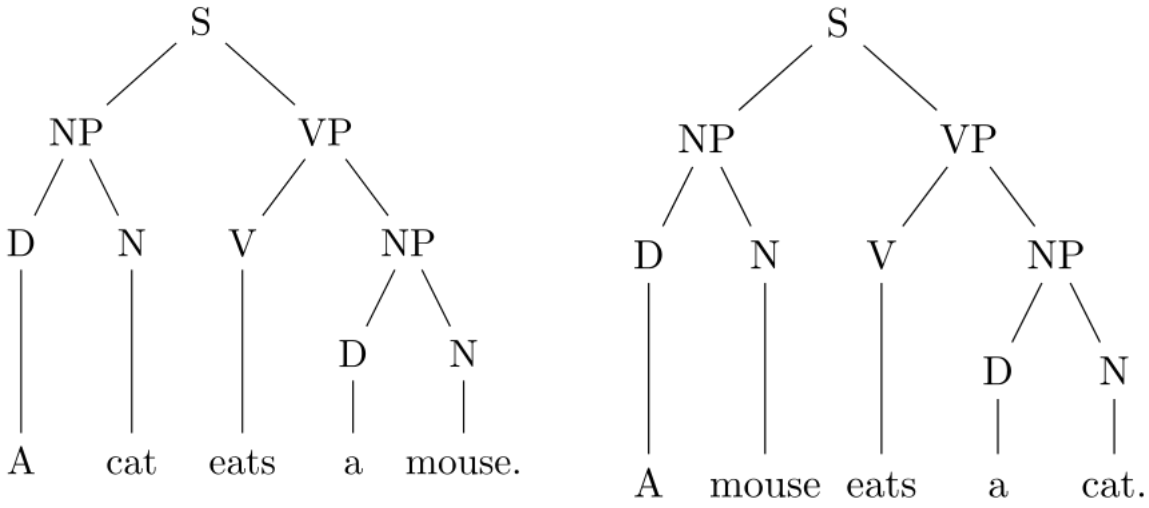
Thus, using well-designed positive-definite kernels will help us compute similarity over trees without explicitly computing the feature vectors of these trees. Kernel methods have also been widely used in ML techniques of support vector machines, and thus there are plenty of algorithms which are currently working natively with kernels, or have an extension that handles kernelization.

Exploration of kernels-

A study on different kinds of kernels should help provide us the kernel providing the best accuracy. The options include-

- ST kernel- (subtree kernel)
- SST kernels- (subset tree kernel)
- QT kernel- (quadratic tree kernel)
- FT kernel - (fast tree kernel)

I will review the kernels' time and accuracy comparing their performance. For example, let us take the following two trees-



ST kernel-

For this kernel, a subtree is defined as a node and all its children. The subtree kernel count the number of common subtrees between two given trees. Here, **the number will be 7**- [NP [D [a]] [N [cat]]],[NP [D [a]] [N [mouse]]],[N [mouse]],[N [cat]],[V [eats]],[D [a]][NP [D [a]] [N [cat]]],[NP [D [a]] [N [mouse]]],[N [mouse]],[N [cat]],[V [eats]],[D [a]] (which is counted twice as it appears twice)

This runs in linear time.

SST Kernel-

This has the same definition as the earlier tree, but is more general in nature. We count the number of common subset trees as well, which will include the ones of the nature VP [V] [NP]. Thus, we will have more number of subset trees here, and **the number will be 54.** Hence, this one will have richer information which may be critical to capture syntactic properties, but also contain too many

irrelevant features, causing overfitting to occur and decreasing the classification accuracy. This runs in linear time.

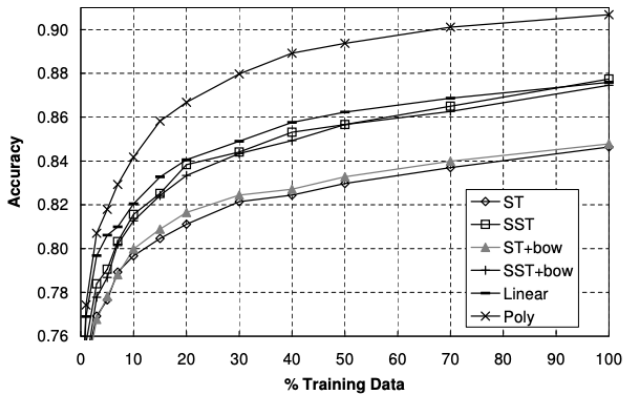
QT Kernel-

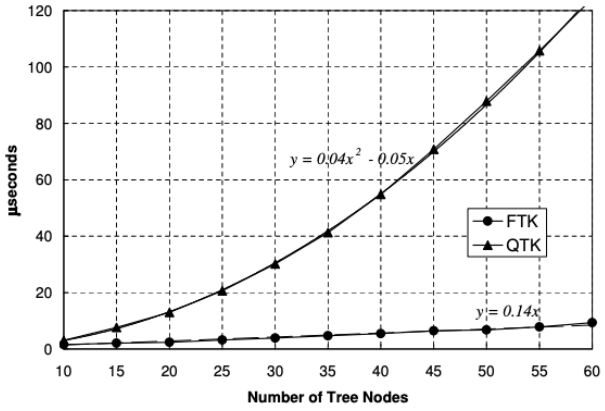
These are an extension of SST Kernels, but are faster.

FT Kernel-

These are another faster version of SST kernel, and are faster than QT kernel

Comparing their performance,





We see that-

- the richer the kernel is in term of substructures (e.g. SST), the higher the accuracy is,
- tree kernels are effective also in case of automatic parse trees
- kernel combinations always improve traditional feature models, the best approach is to combine scalar-based and structured based kernels.

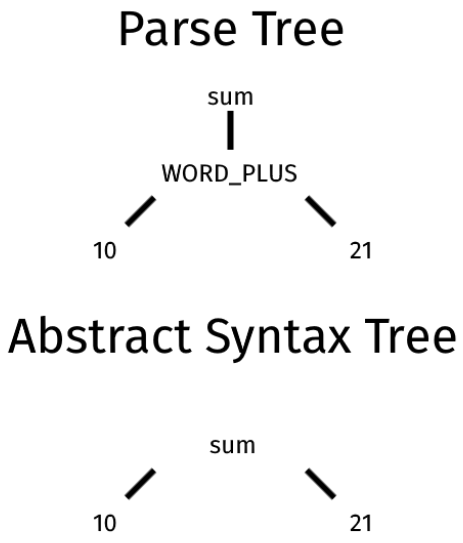
Application to software code-

Extending the above analysis, let us see how this works for software code. Here, we use **Abstract Syntax Tree Kernel**. These kernels have a higher level of abstraction; while a parse tree might contains all the tokens which appeared in the program and possibly a set of intermediate rules. As the abstract syntax trees describe the parse tree logically, it does not need to contain all the syntactical constructs required to parse some source code (like white spaces, braces, keywords, parenthesis, etc). Thus, this is a polished version of the parse tree, in which only the information relevant to understanding the code is maintained, and are thus highly invaluable for program analysis and program transformation systems.

For example, if we have the code-

sum=10 plus 21

this is how Abstract Syntax trees differ from Parse Trees.



From a performance perspective, these perform as well as polynomial FT parsers.

Strategies for ASTs-

There are two strategies for parsing: **top-down parsing** and **bottom-up parsing**. Both terms are defined in relation to the parse tree generated by the parser. In a simple way:

- a top-down parser tries to identity the root of the parse tree first, then it moves down the subtrees, until it find the leaves of the tree.
- a bottom-up parses instead starts from the lowest part of the tree, the leaves, and rise up until it determines the root of the tree.

Traditionally top-down parsers were easier to build, but bottom-up parsers were more powerful.

Algorithms-

I will explore 4 different parsing algorithms and libraries generation here are –

1. ***Cocke–Younger–Kasami-***

   Algorithm-
   Here, we have a dynamic programming algorithm which requires the context-free grammar to be rendered into Chomsky normal form (CNF) and then tests for possibilities to split the current sequence into two smaller sequences, to finally generate a triangular table.

   As an example, look at the grammar-
   S -> AB | BC
   A -> BA | a
   B -> CC | b
   C -> AB | a

   We will get the final triangular table as-

{S, A, C}				
Ø	{S, A, C}			
Ø	{B}	{B}		
{S, A}	{B}	{S, C}	{S, A}	
{B}	{A, C}	{A, C}	{B}	{A, C}

   The worst case running time of CYK is **O(n^3/G)**, where n is the length of the parsed string and G the size of the CNF grammar. There are many open source implementations of this- like CYK Algorithm in Golang and Chart-parsers.

2. *Earley Algorithm-*
   Here, we have a top-down dynamic programming algorithm which uses Earley's recognizer to generate a parse table. This process uses a grammar and parses each input words according to the 3 operations-
   - Predictor: an incomplete entry looks for a symbol to the right of its dot. If there is no matching symbol in the chart, one is predicted by adding all matching rules with an initial dot.
   - Scanner: an incomplete entry looks for a symbol to the right of the dot. this prediction is compared to the input, and a complete entry is added to the chart if it matches.
   - Completer: a complete edge is combined with an incomplete entry that is looking for it to form another complete entry.

   For example, using this grammar,
   P → S (the start rule)
   S → S + M | M
   *M → M * T | T*
   T → number

And input 2 + 3 * 4
We get the following state sets-

state no.	Production	Origin	Comment
S(0): • 2 + 3 * 4			
1	P → • S	0	start rule
2	S → • S + M	0	predict from (1)
3	S → • M	0	predict from (1)
4	M → • M * T	0	predict from (3)
5	M → • T	0	predict from (3)
6	T → • number	0	predict from (5)
S(1): 2 • + 3 * 4			
1	T → number •	0	scan from S(0)(6)
2	M → T •	0	complete from (1) and S(0)(5)
3	M → M • * T	0	complete from (2) and S(0)(4)
4	S → M •	0	complete from (2) and S(0)(3)
5	S → S • + M	0	complete from (4) and S(0)(2)
6	P → S •	0	complete from (4) and S(0)(1)
S(2): 2 + • 3 * 4			
1	S → S + • M	0	scan from S(1)(5)
2	M → • M * T	2	predict from (1)
3	M → • T	2	predict from (1)
4	T → • number	2	predict from (3)
S(3): 2 + 3 • * 4			
1	T → number •	2	scan from S(2)(4)
2	M → T •	2	complete from (1) and S(2)(3)
3	M → M • * T	2	complete from (2) and S(2)(2)
4	S → S + M •	0	complete from (2) and S(2)(1)
5	S → S • + M	0	complete from (4) and S(0)(2)
6	P → S •	0	complete from (4) and S(0)(1)
S(4): 2 + 3 * • 4			
1	M → M * • T	2	scan from S(3)(3)
2	T → • number	4	predict from (1)
S(5): 2 + 3 * 4 •			

1	T → number •	4	scan from S(4)(2)
2	M → M * T •	2	complete from (1) and S(4)(1)
3	M → M • * T	2	complete from (2) and S(2)(2)
4	S → S + M •	0	complete from (2) and S(2)(1)
5	S → S • + M	0	complete from (4) and S(0)(2)
6	P → S •	0	complete from (4) and S(0)(1)

Which gives us the final parse of (P → S •, 0) . These parsers are appealing because they can parse all context-free languages, unlike LR parsers and LL parsers, which can only handle restricted classes of languages. This executes in **O(n^3),** where n is the length of the parsed string. Opensource implementation of this includes Nearley.

3. *LL Algorithm-*

One of the most common algorithms, this uses top-down parsers to parses the input from Left to right, performing Leftmost derivation of the sentence.

An LL parse is a left-to-right, leftmost derivation. That is, we consider the input symbols from the left to the right and attempt to construct a leftmost derivation. This is done by beginning at the start symbol and repeatedly expanding out the leftmost nonterminal until we arrive at the target string. An LR parse is a left-to-right, rightmost derivation, meaning that we scan from the left to right and attempt to construct a rightmost derivation. The parser continuously picks a substring of the input and attempts to reverse it back to a nonterminal.

During an LL parse, the parser continuously chooses between two actions:

- **Predict**: Based on the leftmost nonterminal and some number of lookahead tokens, choose which production ought to be applied to get closer to the input string.
- **Match**: Match the leftmost guessed terminal symbol with the leftmost unconsumed symbol of input.

Notice that in each step we look at the leftmost symbol in our production. If it's a terminal, we match it, and if it's a nonterminal, we predict what it's going to be by choosing one of the rules.

For example, given this grammar:

S → F (the start rule)
S → S + F
F → number(a)

Then given the input- (a+a). we will get the following table

	(	)	a	+	$
S	2	-	1	-	-
F	-	-	3	-	-

Which generates the parser as S → (S + F) → (F + F) → (a + F) → (a + a)

Given the relative simplicity of this algorithm, this is widely used in many opensource libraries to generate parse trees. One of the most common libraires is ANTLR.

4. *LR Algorithm-*

   This is like the LL parser , it is a bottom-up parser which reads input text from left to right and produces a rightmost derivation in reverse At a high level, the difference between LL parsing and LR parsing is that LL parsers begin at the start symbol and try to apply productions to arrive at the target string, whereas LR parsers begin at the target string and try to arrive back at the start symbol. N general, this is not that suitable for Human languages. Some of the open source implementations for this include Jison and Bison .

Summary-

We have examined different types of kernels which can be used for syntactic parsing of documents. We extend the analysis to see how similar trees can be applied in software code using abstract syntax trees. Then we reviewed the strategies for constructing such trees and 4 different algorithms for parsing a sentence to generate the tree.

References-

1. **https://www.semanticscholar.org/paper/Making-Tree-Kernels-Practical-for-Natural-Language-Moschitti/e1f9ad97af9f09b206e3681f080fcf996e56a5e3?p2df**
2. **https://www.cs.tufts.edu/~jfoster/papers/msr05.pdf**
3. **https://www.cs.bgu.ac.il/~michaluz/seminar/CKY1.pdf**
4. **https://archive.org/details/introductiontoau00hopc**
5. **https://arxiv.org/abs/2010.07874**
6. **https://cs.stackexchange.com/questions/43/language-theoretic-comparison-of-ll-and-lr-grammars**