IT3105 Project III: Recognizing Textual Entailment

Part II

Erwin Marsi
IT-VEST 313
emarsi@idi.ntnu.no

Logic and Language Technology Group, Intelligent Systems, IDI, NTNU

Plan for today's lecture

1. Introduction
2. Syntactic analysis
3. Tree edit distance algorithm
4. Break (15 minutes)
5. Tree edit distance algorithm (continued)
6. Normalization & Edit costs
7. Epilog: how to continue from here

Textual entailment

- **Textual entailment** is defined as a directional relation between two text fragments, termed T - the entailing *text*, and H - the entailed *hypothesis*
- T entails H if, typically, a human reading T would infer that H is most likely true

Recognizing Textual Entailment

- **Recognizing Textual Entailment** (RTE) is the task of deciding, given T and H, whether T entails H.
- Carried out fully automatically by a computer program
- Without any human intervention

Beyond lexical matching

- Assumption underlying most RTE approaches:
  if H is similar to T, then entailment is likely
- Requires matching of aligning H to parts of T
- Word matching methods depend on overlap in words, possibly weighted words
- Improvement: add linguistic information such as lemma and/or part-of-speech
- Can we add even more linguistic information?
- Yes, *syntactic information* in the form of *dependency trees*!

Syntactic analysis (1)

- Numerical expression must be well-formed:
  - good: (x + 10) * y
  - bad: * y (x 10 +)
- Likewise English sentences must be well-formed:
  - good: Sookie loves vampire Bill
  - bad: loves Bill Sookie vampire
- Syntactic theory tries to explain which expressions of the language are valid
- This explanation relies syntactic analysis, that is, imposing some structure on top of sentences
- usually as a tree or a more general graph structure

Syntactic analysis (2)

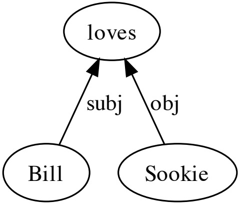
- A syntactic parser is a program that assigns a syntactic structure to an input sentence
- Syntactic parsing: process of assigning syntactic structures to sentences

Dependency structures (1)

- Dependency analysis is a particular form of syntactic analysis
- A dependency parser assigns dependency structures to sentences
- A dependency structure consists of triples:
  - head
  - relation
  - dependent
- Example: *Bill loves Sookie*
  - head: loves
  - relation1: subj (subject)
  - dependent1: Bill
  - relation2: subj (object)
  - dependent2: Sookie

Dependency structures (2)

- Set of dependency triples $< \mathit{dependent}, \mathit{head}, \mathit{relation} >$ defines a dependency tree or graph
- Preprocessed RTE data contains a syntactic structure for each sentence in T and H

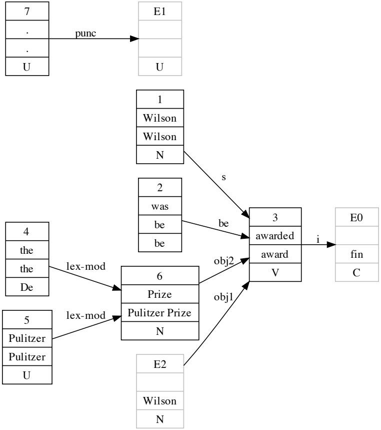


Dependency structures (3)

- Preprocessed RTE data contains a syntactic structure for each sentence in T and H (see example)
- See project description for more information about the XML format
- Implementation details:
  - Problem: sometimes the dependency structure is not fully connected
  - Solution: ignore small trees or connect all trees into one big tree
  - Problem: Text contains multiple sentences
  - Solution: connect them into one big tree

Dependency structures (4)

Wilson was awarded the Pulitzer Prize.



Matching syntactic trees

- Intuition: Text entails Hypothesis if the distance between the distance between their dependency trees is relatively small
- Distance between two trees can be formalized as the number of edit operations required to transform one tree into another
- where edit operations on nodes include insertion, deletionand substitution
- each operation with an associated cost
- Tree edit distance is the overall cost of the transformation
- Therefore T entails H if the (normalized) tree edit distance between T and H is below a certain threshold

Tree edit distance: Preliminary remarks

- The following Section presents the tree edit distance algorithm by Zhang & Shasha
- This is not a course on approximation algorithms, so we skip proofs and complexity analyses, focus on procedure
- Algorithm needs a bit of study: don't worry if you fail to fully understand it here
- Reuses some slides from the excellent presentation by Nikolaus Augsten (see links on project website)

Edit operations: substitute

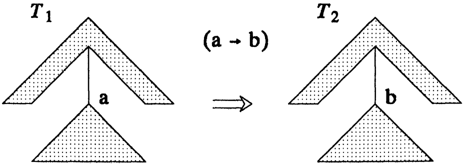


FIG. 1

Edit operations: delete

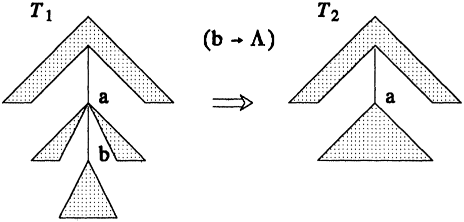


FIG. 2

Edit operations: insert

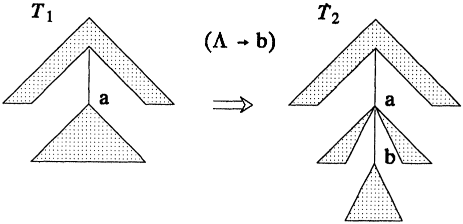


FIG. 3

Edit distance (1)

- Every edit operation has an associated cost γ:
    - substitution cost $\gamma(n \rightarrow m)$
    - deletion cost $\gamma(n \rightarrow \Lambda)$
    - insertion cost $\gamma(\Lambda \rightarrow n)$
- Edit distance between two trees is the sequence of edit operations of which the overall cost in minimal
- By default, assume unit costs (all costs are 1)

Edit distance (2)

- Edit sequence to turn T_1 into T_2
  - delete node c
  - insert node c
- Thus $\mathit{Editdist}(T_1, T_2) = 2$

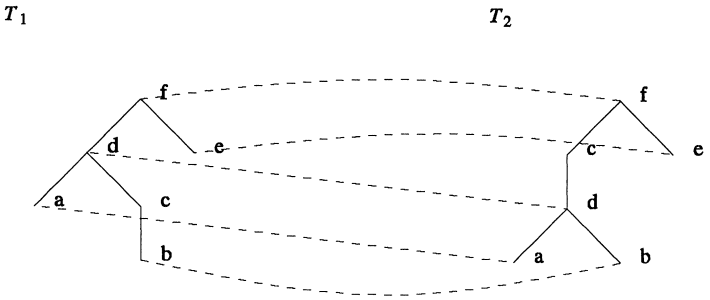
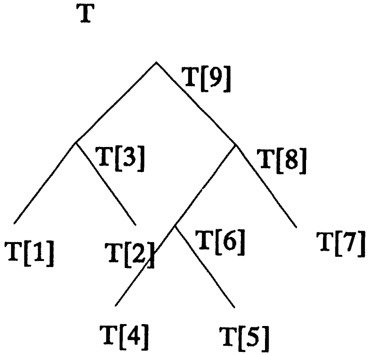


FIG. 4

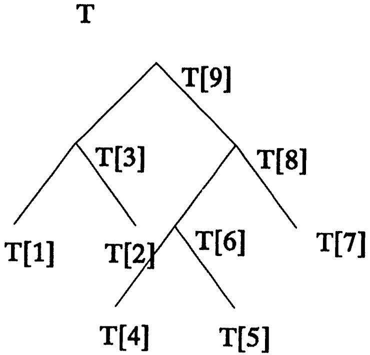
Postorder traveral

- Let $T[i]$ be the ith node in the tree according to the left-to-right postorder numbering



Leftmost leaf descendant

- $l[i]$ is the number of the **leftmost leaf descendant** of the subtree rooted at T[i]



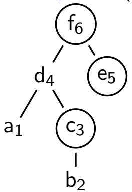
Key Roots

Definition (Key Root)

The set of *key roots* of a tree T is defined as

$$kr(\mathrm{T}) = \{k \in N(\mathrm{T}) \mid \nexists k' \in N(\mathrm{T}) : k' > k \text{ and } l(k) = l(k')\}$$

- Alternative definition: A *key root* is a node of T that either has a left sibling or is the root of T.
- Example: $kr(\mathrm{T}) = \{3, 5, 6\}$



- Only subtrees rooted in a key root need a separate computation.

Forrest

- $T[i..j]$ is the ordered subforest of T induced by the nodes numbered i to j inclusive
- If $i > j$, then $T[i..j] = \varnothing$
- Note that $T[l(i)..i]$ is always a tree
- The distance between $T[i'..i]$ and $T[j'..j]$ is denoted $\mathit{forestdist}(T[i'..i], T[j'..j])$

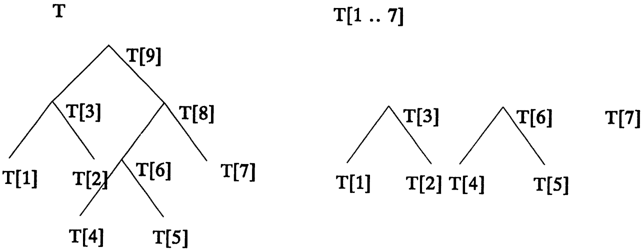
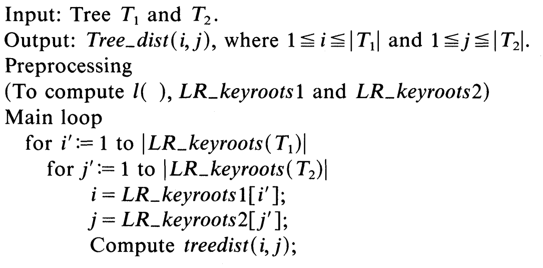


FIG. 5

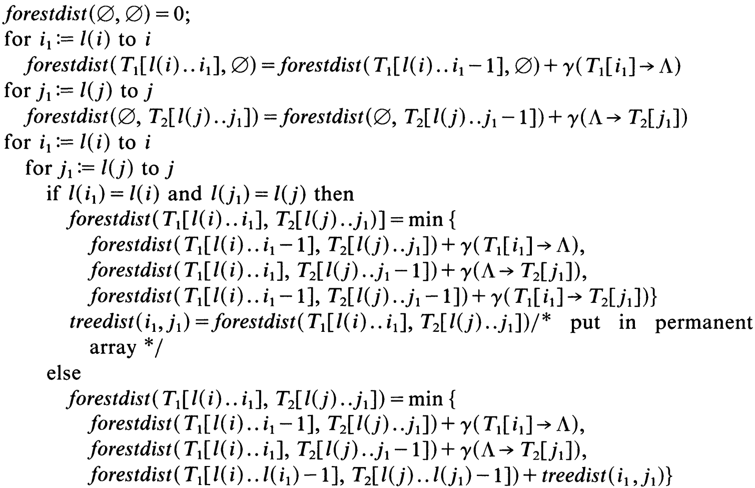
Treedist(T_1, T_2) algorithm



```
Input: Tree $T_1$ and $T_2$.
Output: $Tree\_dist(i, j)$, where $1 \leqq i \leqq |T_1|$ and $1 \leqq j \leqq |T_2|$.
Preprocessing
(To compute $l(\ )$, $LR\_keyroots1$ and $LR\_keyroots2$)
Main loop
  for $i' := 1$ to $|LR\_keyroots(T_1)|$
    for $j' := 1$ to $|LR\_keyroots(T_2)|$
        $i = LR\_keyroots1[i']$;
        $j = LR\_keyroots2[j']$;
        Compute $treedist(i, j)$;
```

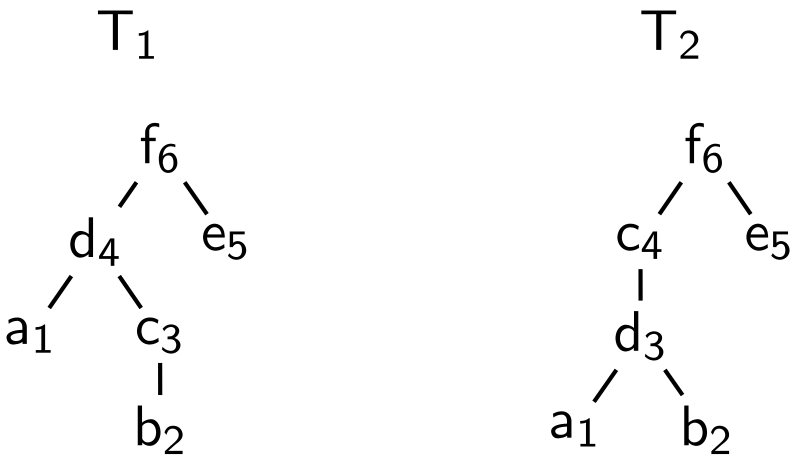
- *treedist* is computed using bottom-up dynamic programming
- forrest distance values are stored in a temporary *forrestdist* array, per treedist computation
- tree distance values are stored in a permanent *treedist* array

$treedist(i, j)$ computation



```
$forestdist(\varnothing, \varnothing) = 0;$
for $i_1 := l(i)$ to $i$
  $forestdist(T_1[l(i)..i_1], \varnothing) = forestdist(T_1[l(i)..i_1 - 1], \varnothing) + \gamma(T_1[i_1] \rightarrow \Lambda)$
for $j_1 := l(j)$ to $j$
  $forestdist(\varnothing, T_2[l(j)..j_1]) = forestdist(\varnothing, T_2[l(j)..j_1 - 1]) + \gamma(\Lambda \rightarrow T_2[j_1])$
for $i_1 := l(i)$ to $i$
  for $j_1 := l(j)$ to $j$
    if $l(i_1) = l(i)$ and $l(j_1) = l(j)$ then
      $forestdist(T_1[l(i)..i_1], T_2[l(j)..j_1)] = \min\{$
        $forestdist(T_1[l(i)..i_1 - 1], T_2[l(j)..j_1]) + \gamma(T_1[i_1] \rightarrow \Lambda),$
        $forestdist(T_1[l(i)..i_1], T_2[l(j)..j_1 - 1]) + \gamma(\Lambda \rightarrow T_2[j_1]),$
        $forestdist(T_1[l(i)..i_1 - 1], T_2[l(j)..j_1 - 1]) + \gamma(T_1[i_1] \rightarrow T_2[j_1])\}$
      $treedist(i_1, j_1) = forestdist(T_1[l(i)..i_1], T_2[l(j)..j_1])$/* put in permanent
        array */
    else
      $forestdist(T_1[l(i)..i_1], T_2[l(j)..j_1]) = \min\{$
        $forestdist(T_1[l(i)..i_1 - 1], T_2[l(j)..j_1]) + \gamma(T_1[i_1] \rightarrow \Lambda),$
        $forestdist(T_1[l(i)..i_1], T_2[l(j)..j_1 - 1]) + \gamma(\Lambda \rightarrow T_2[j_1]),$
        $forestdist(T_1[l(i)..l(i_1) - 1], T_2[l(j)..l(j_1) - 1]) + treedist(i_1, j_1)\}$
```

Example Trees and Edit Costs



- Example: Edit distance between T_1 and T_2.
  - $\omega_{ins} = \omega_{del} = 1$
  - $\omega_{ren} = 0$ for identical rename, otherwise $\omega_{ren} = 1$
- Each of the following slide is the result of a call of forest-dist().

Executing the Algorithm (1/9)

	1	2	3	4	5	6
l_1	1	2	2	1	5	1

	1	2	3
kr_1	3	5	6
	x ↑		

	1	2	3	4	5	6
l_2	1	2	1	1	5	1

	1	2	3
kr_2	2	5	6
	y ↑		

- $i = kr_1[x] = 3 \Rightarrow l_1[i] = 2$
- $j = kr_2[y] = 2 \Rightarrow l_2[j] = 2$

- temporary array *fd*:

$\text{d}_j \rightarrow$		2
$\text{d}_i \downarrow$	0	1
2	1	0
3	2	1

▨ $l_1[i] = l_1[\text{d}_i]$ and $l_2[j] = l_2[\text{d}_j]$

- permanent array *td*:

	1	2	3	4	5	6
1						
2		0				
3		1				
4						
5						
6						

Executing the Algorithm (2/9)

	1	2	3	4	5	6
l_1	1	2	2	1	5	1

	1	2	3
kr_1	3	5	6
	x↑		

	1	2	3	4	5	6
l_2	1	2	1	1	5	1

	1	2	3
kr_2	2	5	6
		y↑	

- $i = kr_1[x] = 3 \Rightarrow l_1[i] = 2$
- $j = kr_2[y] = 5 \Rightarrow l_2[j] = 5$

- temporary array *fd*:

	$\mathrm{d}_j \rightarrow$	5
$\mathrm{d}_i \downarrow$	0	1
2	1	1
3	2	2

▨ $l_1[i] = l_1[\mathrm{d}_i]$ and $l_2[j] = l_2[\mathrm{d}_j]$

- permanent array *td*:

	1	2	3	4	5	6
1						
2		0			1	
3		1			2	
4						
5						
6						

Executing the Algorithm (3/9)

	1	2	3	4	5	6
l_1	1	2	2	1	5	1

	1	2	3
kr_1	3	5	6
	x ↑		

	1	2	3	4	5	6
l_2	1	2	1	1	5	1

	1	2	3
kr_2	2	5	6
			y ↑

- $i = kr_1[x] = 3 \Rightarrow l_1[i] = 2$
- $j = kr_2[y] = 6 \Rightarrow l_2[j] = 1$

- temporary array *fd*:

	$\mathrm{d}_j \rightarrow$	1	2	3	4	5	6
$\mathrm{d}_i \downarrow$	0	1	2	3	4	5	6
2	1	1	1	2	3	4	5
3	2	2	2	2	2	3	4

▨ $l_1[i] = l_1[\mathrm{d}_i]$ and $l_2[j] = l_2[\mathrm{d}_j]$

- permanent array *td*:

	1	2	3	4	5	6
1						
2	1	0	2	3	1	5
3	2	1	2	2	2	4
4						
5						
6						

Executing the Algorithm (4/9)

	1	2	3	4	5	6
l_1	1	2	2	1	5	1

	1	2	3
kr_1	3	5	6
		x ↑	

	1	2	3	4	5	6
l_2	1	2	1	1	5	1

	1	2	3
kr_2	2	5	6
	y ↑		

- $i = kr_1[x] = 5 \Rightarrow l_1[i] = 5$
- $j = kr_2[y] = 2 \Rightarrow l_2[j] = 2$

- temporary array *fd*:

$\mathrm{d}_i \downarrow$ \ $\mathrm{d}_j \rightarrow$		2
	0	1
5	1	1

🟦 $l_1[i] = l_1[\mathrm{d}_i]$ and $l_2[j] = l_2[\mathrm{d}_j]$

- permanent array *td*:

	1	2	3	4	5	6
1						
2	1	0	2	3	1	5
3	2	1	2	2	2	4
4						
5		1				
6						

Executing the Algorithm (5/9)

	1	2	3	4	5	6
l_1	1	2	2	1	5	1

	1	2	3
kr_1	3	5	6
		x ↑	

	1	2	3	4	5	6
l_2	1	2	1	1	5	1

	1	2	3
kr_2	2	5	6
		y ↑	

- $i = kr_1[x] = 5 \Rightarrow l_1[i] = 5$
- $j = kr_2[y] = 5 \Rightarrow l_2[j] = 5$

- temporary array *fd*:

$\mathrm{d}_j \rightarrow$		5
$\mathrm{d}_i \downarrow$	0	1
5	1	0

▨ $l_1[i] = l_1[\mathrm{d}_i]$ and $l_2[j] = l_2[\mathrm{d}_j]$

- permanent array *td*:

	1	2	3	4	5	6
1						
2	1	0	2	3	1	5
3	2	1	2	2	2	4
4						
5		1			0	
6						

Executing the Algorithm (6/9)

	1	2	3	4	5	6
l_1	1	2	2	1	5	1

	1	2	3
kr_1	3	5	6
		x↑	

	1	2	3	4	5	6
l_2	1	2	1	1	5	1

	1	2	3
kr_2	2	5	6
			y↑

- $i = kr_1[x] = 5 \Rightarrow l_1[i] = 5$
- $j = kr_2[y] = 6 \Rightarrow l_2[j] = 1$

- temporary array *fd*:

	$d_j \rightarrow$	1	2	3	4	5	6
$d_i \downarrow$	0	1	2	3	4	5	6
5	1	1	2	3	4	4	5

▨ $l_1[i] = l_1[d_i]$ and $l_2[j] = l_2[d_j]$

- permanent array *td*:

	1	2	3	4	5	6
1						
2	1	0	2	3	1	5
3	2	1	2	2	2	4
4						
5	1	1	3	4	0	5
6						

Executing the Algorithm (7/9)

	1	2	3	4	5	6
l_1	1	2	2	1	5	1

	1	2	3
kr_1	3	5	6
			x ↑

	1	2	3	4	5	6
l_2	1	2	1	1	5	1

	1	2	3
kr_2	2	5	6
	y ↑		

- $i = kr_1[x] = 6 \Rightarrow l_1[i] = 1$
- $j = kr_2[y] = 2 \Rightarrow l_2[j] = 2$

- temporary array *fd*:

$\mathrm{d}_j \rightarrow$		2
$\mathrm{d}_i \downarrow$	0	1
1	1	1
2	2	1
3	3	2
4	4	3
5	5	4
6	6	5

▨ $l_1[i] = l_1[\mathrm{d}_i]$ and $l_2[j] = l_2[\mathrm{d}_j]$

- permanent array *td*:

	1	2	3	4	5	6
1		1				
2	1	0	2	3	1	5
3	2	1	2	2	2	4
4		3				
5	1	1	3	4	0	5
6		5				

Executing the Algorithm (8/9)

	1	2	3	4	5	6
l_1	1	2	2	1	5	1

	1	2	3
kr_1	3	5	6
			x ↑

	1	2	3	4	5	6
l_2	1	2	1	1	5	1

	1	2	3
kr_2	2	5	6
	y ↑		

- $i = kr_1[x] = 6 \Rightarrow l_1[i] = 1$
- $j = kr_2[y] = 5 \Rightarrow l_2[j] = 5$

- temporary array *fd*:

$\text{d}_i \downarrow$ \ $\text{d}_j \rightarrow$		5
	0	1
1	1	▨ 1
2	2	2
3	3	3
4	4	▨ 4
5	5	4
6	6	▨ 5

▨ $l_1[i] = l_1[\text{d}_i]$ and $l_2[j] = l_2[\text{d}_j]$

- permanent array *td*:

	1	2	3	4	5	6
1		1			▨ 1	
2	1	0	2	3	1	5
3	2	1	2	2	2	4
4		3			▨ 4	
5	1	1	3	4	0	5
6		5			▨ 5	

Executing the Algorithm (9/9)

l_1	1	2	3	4	5	6
	1	2	2	1	5	1

kr_1	1	2	3
	3	5	6
			x↑

l_2	1	2	3	4	5	6
	1	2	1	1	5	1

kr_2	1	2	3
	2	5	6
			y↑

- $i = kr_1[x] = 6 \Rightarrow l_1[i] = 1$
- $j = kr_2[y] = 6 \Rightarrow l_2[j] = 1$

- temporary array *fd*:

	$\mathrm{d}_j \rightarrow$	1	2	3	4	5	6
$\mathrm{d}_i \downarrow$	0	1	2	3	4	5	6
1	1	0	1	2	3	4	5
2	2	1	0	1	2	3	4
3	3	2	1	2	3	4	5
4	4	3	2	1	2	3	4
5	5	4	3	2	3	2	3
6	6	5	4	3	3	3	2

🟦 $l_1[i] = l_1[\mathrm{d}_i]$ and $l_2[j] = l_2[\mathrm{d}_j]$

- permanent array *td*:

	1	2	3	4	5	6
1	0	1	2	3	1	5
2	1	0	2	3	1	5
3	2	1	2	2	2	4
4	3	3	1	2	4	4
5	1	1	3	4	0	5
6	5	5	3	3	5	2

Normalization (II-b)

1. Problem: absolute edit distance depends on sentence length
2. Recall that Word Match score was normalized by dividing by total no of words in H
3. Similarly, Edit Distance can be normalized by dividing by the costs of inserting the whole tree H into T

Edit costs (II-c)

1. When calculating editdist(T,H) it makes sense to set the cost for deletion to 0, because T may contain additional irrelevant info
2. Insertion and substitution costs are set to 1
3. Insertion cost can be improved by taking the IDF weight of the inserted word

How to go on from here

1. Read the project description for Part II
2. Read the second part of the lecture notes
3. Study relevant parts (only 6 out of 18 pages) of Zhang & Shasha paper: see the annotated version `zs_annot.pdf` available from website
4. Download and study partial implementation in Python: see `tree_edit_dist_incomplete.py` available from website
5. Fill in the missing parts with your own code (possibly recode to your language of choice)
6. Implement extracting trees from preprocessed RTE data and computing edit distance
7. Continue with the other subtasks of Part II of the project
8. Next lecture is in room F6 on Friday, 11th on november, 8.00-10.00 hrs