Dependency Parsing

Guest lecture in Computational Linguistics course

Barbara Plank
bplank.github.io

11-10-2018

Why Parsing?

Why Parsing?

For Whom?

For Whom?

- Researchers working on syntax or related topics within other traditions
- Researchers and application developers interested in using parsers as components in larger systems

Two views of grammatical structure

- So far (a.o.): Constituency structure (a.k.a. phrase structure - CFGs)
- Today: Dependency structure

Today

1. Introduction
2. Transition-based parsing
3. Evaluation

Outline

1. Introduction
   - Introduction to UD
2. Transition-based parsing
3. Evaluation

The notion of dependency

In a dependency grammar, syntactic structures consist of *words* that are linked pairwise by relations called *dependencies*.

The following slides are based on a tutorial by J.Nivre et al: `http://universaldependencies.org/eacl17tutorial/intro.pdf`

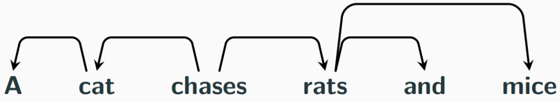
Introduction to UD

Introduction

- Increasing interest in multilingual NLP
  - Multilingual evaluation campaigns to test generality
  - Cross-lingual learning to support low-resource languages
- Increasing awareness of methodological problems
  - Current NLP relies heavily on annotation
  - Annotation schemes vary across languages

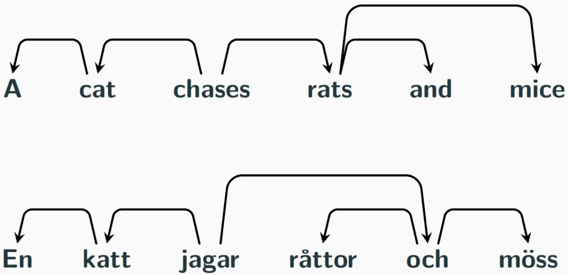
Introduction to UD

Introduction



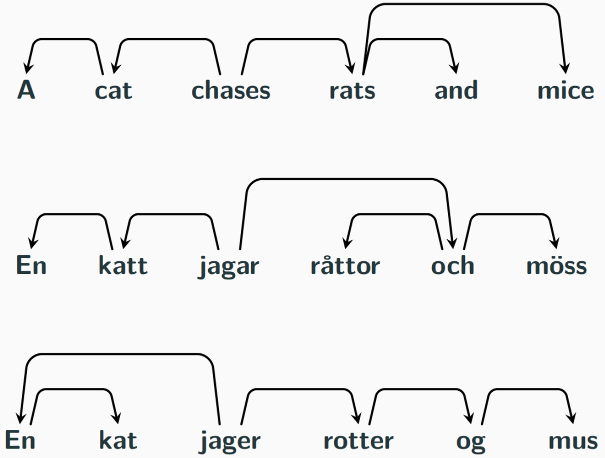
Introduction to UD

Introduction



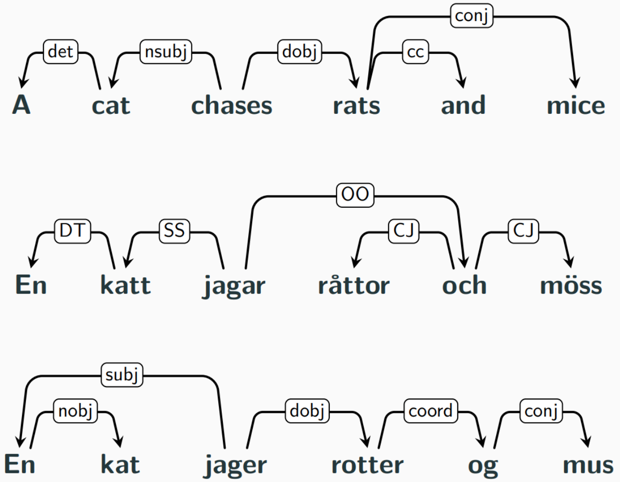
Introduction to UD

Introduction



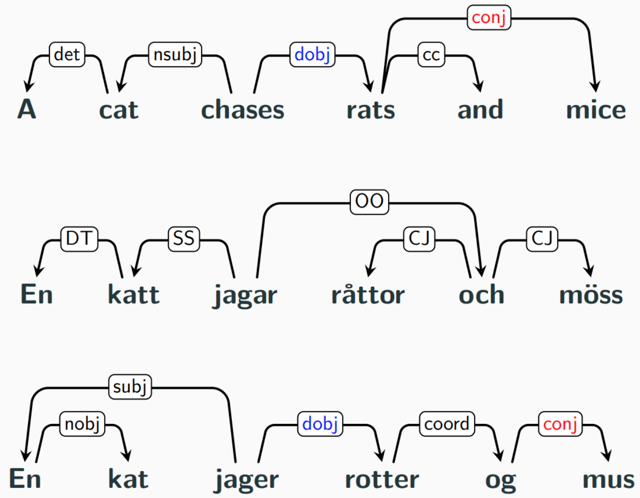
Introduction to UD

Introduction



Introduction to UD

Introduction



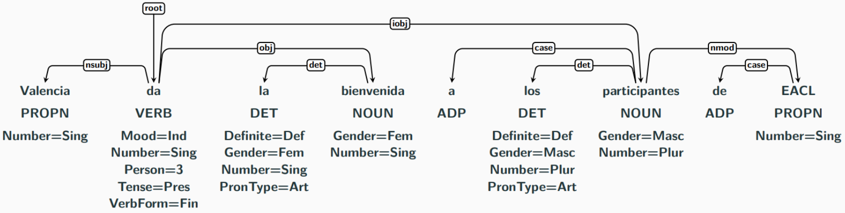
Why is this a problem?

- Hard to compare empirical results across languages
- Hard to usefully do cross-lingual structure transfer
- Hard to evaluate cross-lingual learning
- Hard to build and maintain multilingual systems
- Hard to make comparative linguistic studies
- Hard to validate linguistic typology
- Hard to make progress towards a universal parser

Introduction to UD

Introduction

http://universaldependencies.org



- Part-of-speech tags
- Morphological features
- Syntactic dependencies

Goals and Requirements

- Cross-linguistically consistent grammatical annotation
- Support multilingual NLP and linguistic research
- Build on common usage and existing de facto standards
- Complement – not replace – language-specific schemes
- Open community effort – anyone can contribute!

The UD Philosophy

- Maximize parallelism – but don't overdo it
  - Don't annotate the same thing in different ways
  - Don't make different things look the same
  - Don't annotate things that are not there
- Universal taxonomy with language-specific elaboration
  - Languages select from a universal pool of categories
  - Allow language-specific extensions

Morphological Annotation

Le	chat	chasse	les	chiens	.
le	chat	chasser	le	chien	.
DET	NOUN	VERB	DET	NOUN	PUNCT
Definite=Def Gender=Masc Number=Sing	Gender=Masc Number=Sing	Mood=Ind Number=Sing Person=3 Tense=Pres VerbForm=Fin	Definite=Def Gender=Masc Number=Plur	Gender=Masc Number=Plur	

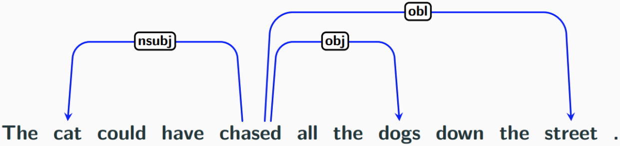
- Lemma representing the semantic content of a word
- Part-of-speech tag representing its grammatical class
- Features representing lexical and grammatical properties of the lemma or the particular word form

Syntactic Annotation

The cat could have chased all the dogs down the street .

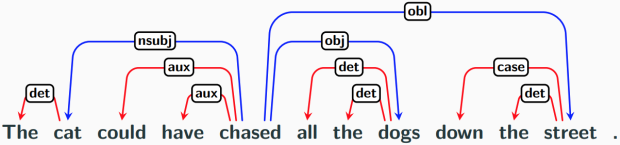
- Content words are related by dependency relations
- Function words attach to the content word they modify
- Punctuation attach to head of phrase or clause

Syntactic Annotation



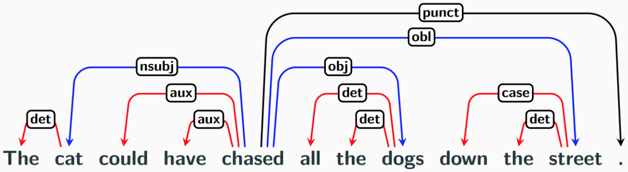
- Content words are related by dependency relations
- Function words attach to the content word they modify
- Punctuation attach to head of phrase or clause

Syntactic Annotation



- Content words are related by dependency relations
- Function words attach to the content word they modify
- Punctuation attach to head of phrase or clause

Syntactic Annotation



- Content words are related by dependency relations
- Function words attach to the content word they modify
- Punctuation attach to head of phrase or clause

CoNLL-U Format

ID	FORM	LEMMA	UPOSTAG	XPOSTAG	FEATS	HEAD	DEPREL	DEPS	MISC
1	Le	le	DET	_	_	2	det	_	_
2	chat	chat	NOUN	_	_	3	nsubj	_	_
3	boit	boire	VERB	_	_	0	root	_	_
4-5	du	_	_	_	_	_	_	_	_
4	de	de	ADP	_	_	6	case	_	_
5	le	le	DET	_	_	6	det	_	_
6	lait	lait	NOUN	_	_	3	obj	_	SpaceAfter=No
7	.	.	PUNCT	_	_	3	punct	_	_

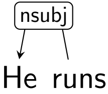
- Revised and extended version of CoNLL-X format
- Two-level segmentation and enhanced dependencies

Naming nodes in a dependency

`rel(head,dep)`

- head vs dependent
- governor vs modifier
- regent vs subordinate
- parent vs child

In the convention we use, dependency edges go from head to dependent: `nsubj(runs,He)`.

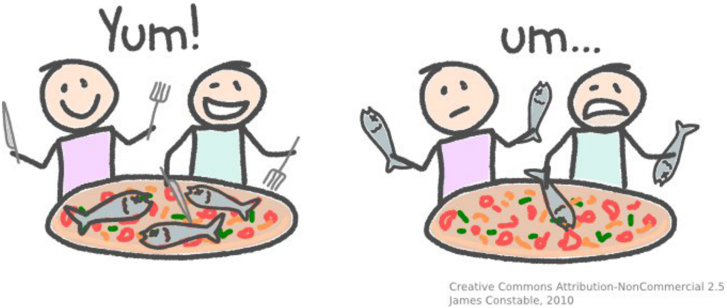


Example

"They ate the pizza with anchovies"

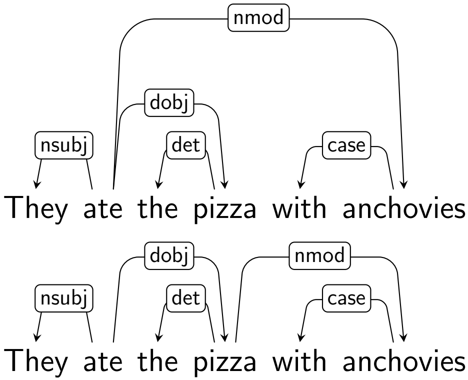
Example

"They ate the pizza with anchovies"



Dependency Trees: Universal dependencies

"They ate the pizza with anchovies"



Outline

1. Introduction

2. Transition-based parsing
   - Choosing the Right Actions
   - Representing configurations using feature templates

3. Evaluation

What is Transition-based Parsing?

- One of the two leading approaches for dependency parsing

What is Transition-based Parsing?

- One of the two leading approaches for dependency parsing
  - Approach 1: **Transition-based parsing:** local decisions
  - Approach 2: **Graph-based parsing:** global decision (find globally best tree; computationally more expensive; we will not cover this)

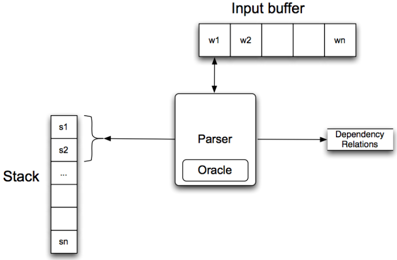
Why Transition-based Parsing?

- left to right: similar to how the human brain does it
- in recent years: state-of-the-art accuracy
- very fast
- simple
- flexible: also suitable for producing phrase-structure trees, CCG derivations, semantic representations... (see next lectures)

Transition-based Parsing: How? Intuition:

- Read sentence word by word, left to right
- Build up the dependency tree one word at a time:
  - after each word, look at the current **parser configuration**
  - select a **parser operation** from a set of operations consulting a machine-learned classifier

What is a parser configuration?



Configuration:

- Buffer B (words left, at the start entire sentence)
- Stack S (last in, first out)
- Relations R (dependency edges predicted so far, a partial parse)

Transition-based parsing

- Configuration:
  - Buffer B (words left, at the start entire sentence)
  - Stack S (last in, first out)
  - Relations R (dependency edges predicted so far, a partial parse)
- Configuration $C = S, B, R$

Transition-based parsing

- Configuration:
  - Buffer B (words left, at the start entire sentence)
  - Stack S (last in, first out)
  - Relations R (dependency edges predicted so far, a partial parse)
- Configuration $C = S, B, R$

- Initial configuration: empty stack, all words on buffer, empty R
- Final configuration: stack, empty buffer, all edges are in R

Transition-based parsing

- Configuration:
  - Buffer B (words left, at the start entire sentence)
  - Stack S (last in, first out)
  - Relations R (dependency edges predicted so far, a partial parse)
- Configuration $C = S, B, R$

- Initial configuration: empty stack, all words on buffer, empty R
- Final configuration: stack, empty buffer, all edges are in R
- Parser does a search through the space of possible configurations

Basic actions (simplified)

- The parser has 3 basic operations (other variants possible):

Basic actions (simplified)

- The parser has 3 basic operations (other variants possible):
  - **Shift:** Move a word from the buffer to the stack
  - **Left:** Create an edge to the left
  - **Right:** Create an edge to the right
- (This transition system with 3 operations is called *arc-standard*)

Basic actions (details)

- The parser has 3 basic operations (other variants possible):
  - **Shift:** Move a word from the buffer to the stack
    $(S, i|j|B, A) \rightarrow (S|i, j|B, A)$
  - **Left:** Create an edge to the left
    $(S|i|j, B, A) \rightarrow (S|j, B, A \cup j \rightarrow i)$ [create an edge from j to i, where j is the first and i the second node from the top of the stack; in addition removes i from stack)]
  - **Right:** Create an edge to the right
    $(S|i|j, B, A) \rightarrow (S|i, B, A \cup i \rightarrow j)$ [create an edge from i to j, where i is the second and j the first node on top of the stack; pops j from the stack]
- Details in `http://stp.lingfil.uu.se/~nivre/master/transition.pdf`

Transition-based Dependency Parsing

```
buffer = ['They', 'ate', 'the', 'pizza',
           'with', anchovies']
stack = []
```

Transition-based Dependency Parsing

```
buffer = ['They', 'ate', 'the', 'pizza',
           'with', anchovies']
stack = []


while len(buffer) > 0 or len(stack) > 1:
    action = choose_action(buffer, stack)
    if action == 'SHIFT':
        stack.append(i)
    elif action == 'LEFT':
       parse.add(stack[-2], stack.pop())
    elif action == 'RIGHT':
       parse.add(i, stack.pop())
```

Example

Stack: [] Buffer: [They, ate, the, pizza, with, anchovies]
R: []

Example

Stack: [They] Buffer: [ate, the, pizza, with, anchovies]
R: []

SHIFT,

Example

Stack: [They, ate]   Buffer: [the, pizza, with, anchovies]
R: []

SHIFT,SHIFT

Example

Stack: [ate]   Buffer: [the, pizza, with, anchovies]
R: [ate $\rightarrow$ They]

SHIFT,**SHIFT**,**Left**,

Example

Stack: [ate, the] Buffer: [pizza, with, anchovies]
R: [ate $\rightarrow$ They]

SHIFT,**SHIFT**,**Left**,**SHIFT**,

Example

Stack: [ate, the, pizza]   Buffer: [with, anchovies]
R: [ate $\rightarrow$ They]

SHIFT,**SHIFT**,**Left**,**SHIFT**,**SHIFT**,

Example

Stack: [ate, pizza] Buffer: [with, anchovies]
R: [ate $\rightarrow$ They, pizza $\rightarrow$ the]

SHIFT,**SHIFT**,**Left**,**SHIFT**,**SHIFT**,**Left**,

Example

Stack: [ate, pizza, with]   Buffer: [anchovies]
R: [ate $\rightarrow$ They, pizza $\rightarrow$ the]

SHIFT,SHIFT,Left,SHIFT,SHIFT,Left,SHIFT,

Example

Stack: [ate, pizza, with,anchovies]   Buffer: []
R: [ate $\rightarrow$ They, pizza $\rightarrow$ the]

SHIFT,SHIFT,Left,SHIFT,SHIFT,Left,SHIFT,SHIFT,

Example

Stack: [ate, pizza,anchovies] Buffer: []
R: [ate $\rightarrow$ They, pizza $\rightarrow$ the, anchovies $\rightarrow$ with]

SHIFT,SHIFT,Left,SHIFT,SHIFT,Left,SHIFT,SHIFT,Left

Example

Stack: [ate, pizza]   Buffer: []
R: [ate $\rightarrow$ They, pizza $\rightarrow$ the, anchovies $\rightarrow$ with, pizza $\rightarrow$ anchovies]

SHIFT,**SHIFT**,**Left**,**SHIFT**,**SHIFT**,**Left**,**SHIFT**,**SHIFT**,**Left**,
Right

Example

Stack: [ate]   Buffer: []

R: [ate $\rightarrow$ They, pizza $\rightarrow$ the, anchovies $\rightarrow$ with, pizza $\rightarrow$ anchovies, ate $\rightarrow$ pizza]

SHIFT,**SHIFT**,**Left**,**SHIFT**,**SHIFT**,**Left**,**SHIFT**,**SHIFT**,**Left**, **Right**,**Right**

Example

Stack: []   Buffer: []

R: [ate $\rightarrow$ They, pizza $\rightarrow$ the, anchovies $\rightarrow$ with, pizza $\rightarrow$ anchovies, ate $\rightarrow$ pizza, ROOT$\rightarrow$ ate]

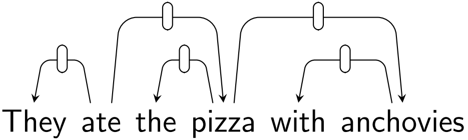
SHIFT,**SHIFT**,**Left**,**SHIFT**,**SHIFT**,**Left**,**SHIFT**,**SHIFT**,**Left**, **Right**,**Right**,**Right**

Example

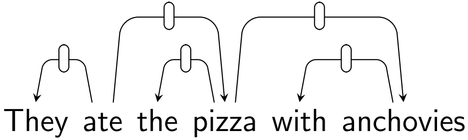
Stack: []   Buffer: []
R: [ate → They, pizza → the, anchovies → with, pizza → anchovies, ate → pizza, ROOT→ ate]

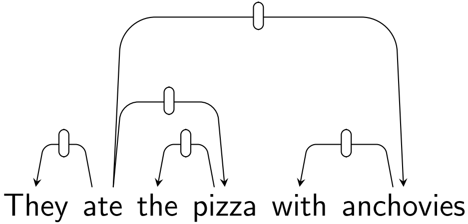
SHIFT,**SHIFT**,**Left**,**SHIFT**,**SHIFT**,**Left**,**SHIFT**,**SHIFT**,**Left**, **Right**,**Right**,**Right**

R encodes our dependency tree:

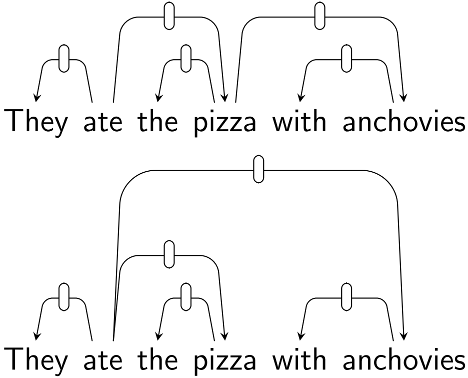


How would we get the other parse tree?





How would we get the other parse tree?



Insight: each sequence of operations derives a dependency tree

Back to our example - alternative

Stack: [ate, pizza]   Buffer: [with, anchovies]
R: [ate $\rightarrow$ They, pizza $\rightarrow$ the]

SHIFT,**SHIFT**,**Left**,**SHIFT**,**SHIFT**,**Left**,

Back to our example - alternative

Stack: [ate]   Buffer: [with, anchovies]
R: [ate $\rightarrow$ They, pizza $\rightarrow$ the, ate $\rightarrow$ pizza]

SHIFT,**SHIFT**,**Left**,**SHIFT**,**SHIFT**,**Left**,**Right**,

Back to our example - alternative

Stack: [ate,with]   Buffer: [anchovies]
R: [ate → They, pizza → the, ate → pizza]

SHIFT,**SHIFT**,**Left**,**SHIFT**,**SHIFT**,**Left**,**Right**,**SHIFT**,

Back to our example - alternative

Stack: [ate,with,anchovies]   Buffer: []
R: [ate $\rightarrow$ They, pizza $\rightarrow$ the, ate $\rightarrow$ pizza]

SHIFT,**SHIFT**,**Left**,**SHIFT**,**SHIFT**,**Left**,**Right**,**SHIFT**, **SHIFT**,

Back to our example - alternative

Stack: [ate,anchovies]   Buffer: []
R: [ate $\rightarrow$ They, pizza $\rightarrow$ the, ate $\rightarrow$ pizza, anchovies $\rightarrow$ with]

SHIFT,**SHIFT**,**Left**,**SHIFT**,**SHIFT**,**Left**,**Right**,**SHIFT**, **SHIFT**, **Left**,

Back to our example - alternative

Stack: [ate]   Buffer: []
R: [ate $\rightarrow$ They, pizza $\rightarrow$ the, ate $\rightarrow$ pizza, anchovies $\rightarrow$ with, ate $\rightarrow$ anchovies]

SHIFT,**SHIFT**,**Left**,**SHIFT**,**SHIFT**,**Left**,**Right**,**SHIFT**, **SHIFT**, **Left**,**Right**

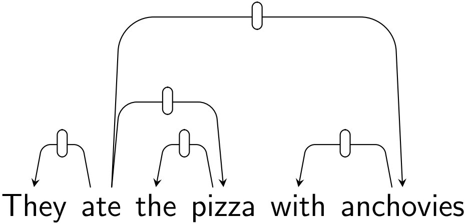
Back to our example - alternative

Stack: [] Buffer: []
R: [ate $\rightarrow$ They, pizza $\rightarrow$ the, ate $\rightarrow$ pizza, anchovies $\rightarrow$ with, ate $\rightarrow$ anchovies, ROOT $\rightarrow$ ate]

SHIFT,**SHIFT**,**Left**,**SHIFT**,**SHIFT**,**Left**,**Right**,**SHIFT**, **SHIFT**, **Left**,**Right**,**Right**

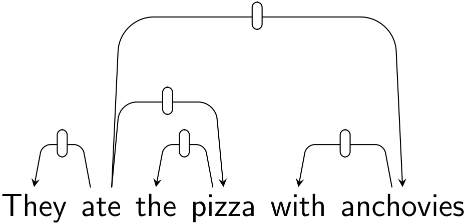
Back to our example - alternative

SHIFT,SHIFT,Left,SHIFT,SHIFT,Left,Right,SHIFT, SHIFT, Left,Right,Right

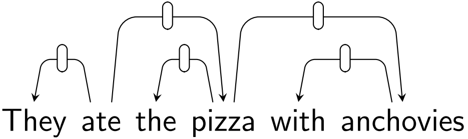


Back to our example - alternative

SHIFT,**SHIFT**,**Left**,**SHIFT**,**SHIFT**,**Left**,**Right**,**SHIFT**, **SHIFT**, **Left**,**Right**,**Right**



SHIFT,**SHIFT**,**Left**,**SHIFT**,**SHIFT**,**Left**,**SHIFT**,**SHIFT**,**Left**, **Right**,**Right**,**Right**



Which Action to Choose?

```
def choose_action(stack, buffer):
    # ???
```

Thanks to Kilian Evang for the basis of the following slides.

Which Action to Choose?

stack:

ate the pizza

They

buffer:

with anchovies

Which Action to Choose?

stack:

ate the pizza

They

buffer:

with anchovies

Next action should be **LEFT**. But how does the parser know that?

Look at Contextual Clues

E.g. word unigram features

stack:

ate the pizza

They

buffer:

with anchovies

Look at Contextual Clues

E.g. word unigram features

stack:

ate the pizza

They

buffer:

with anchovies

Describe configuration in terms of features

```
's_w0=pizza'  # word on top of stack
```

Look at Contextual Clues

E.g. word unigram features

stack:

ate the pizza

They

buffer:

with anchovies

Look at Contextual Clues

E.g. word unigram features

stack: **buffer:**

ate the [pizza] [with] anchovies

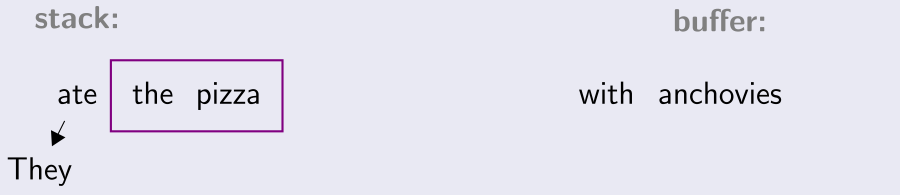
They

Describe configuration in terms of features

```
's_w0=pizza'  # word on top of stack
's_p0=NOUN'  # pos tag on top of stack
'b_w0=with' # first word on buffer
```

Look at Contextual Clues - they get weights

E.g. part-of-speech bigram features



Look at Contextual Clues - they get weights

E.g. part-of-speech bigram features

stack: **buffer:**

ate [the pizza] with anchovies

They (dependent of ate)

Look up "weights" for each possible action

```
weight['s_p1=DET;s_p0=NOUN']['SHIFT'] = -3
weight['s_p1=DET;s_p0=NOUN']['LEFT'] = 10
weight['s_p1=DET;s_p0=NOUN]['RIGHT'] = -5
```

Look at Contextual Clues - they get weights

E.g. word unigram features

stack:

ate the pizza

They

buffer:

with anchovies

Look at Contextual Clues - they get weights

E.g. word unigram features

stack:

ate the pizza

They

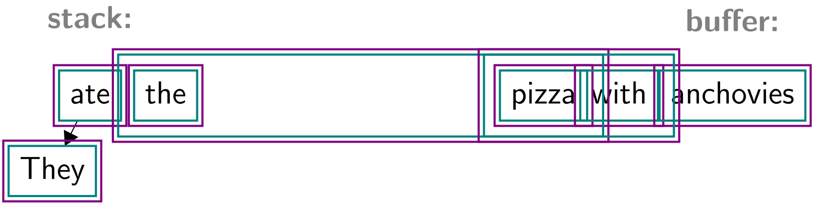
buffer:

with anchovies

Look up "weights" for each possible action

```
weight['s_w1=the']['SHIFT'] = 5
weight['s_w1=the']['LEFT'] = 5
weight['s_w1=the']['RIGHT'] = -5
```

In Practice: Many More Features



Sum up the weights for each possible action, choose the action with the highest total.

Where Do the Weights Come from?

- need training data = sentences where correct actions are known
- training = automatically find weights that lead to good parses
- e.g. perceptron training

Perceptron Training

- start with all weights = 0
- parse the training data
- whenever the parser chooses the wrong action,
  - subtract 1 from the context weights for this action
  - add 1 to the context weights for the correct action
- over time, parser makes fewer mistakes

Perceptron Training: Example

stack:

ate the pizza

They

buffer:

with anchovies

E.g. **LEFT** is correct, parser chooses **SHIFT**.

Perceptron Training: Example

stack: ate [[the] pizza] ← They

buffer: with anchovies

E.g. **LEFT** is correct, parser chooses **SHIFT**.
Update:

```
weight['s_w1=the']['SHIFT'] -= 1
weight['s_w1=the']['LEFT'] += 1

weight['s_p1=DET;s_p0=NOUN']['SHIFT'] -= 1
weight['s_p1=DET;s_p0=NOUN']['LEFT'] += 1
```

Choosing the Right Actions

Summary

- goal: automatically find syntactic structure
- process sentences one word at a time
- at each step, choose the right action
- train parser using training data, features, perceptron training
- simple and works well in practice

Features - Example configuration:

`United canceled the morning flights to Houston`

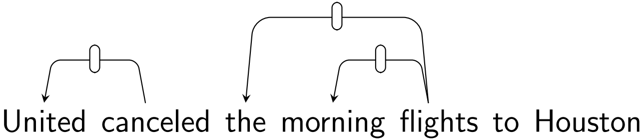
Stack: [root, canceled, flights]   Buffer: [to, Houston]
R: [canceled $\rightarrow$ United, flights $\rightarrow$ morning, flights $\rightarrow$ the]

Features - Example configuration:

`United canceled the morning flights to Houston`

Stack: [root, canceled, flights] Buffer: [to, Houston]
R: [canceled → United, flights → morning, flights → the]

Partial structure so far:

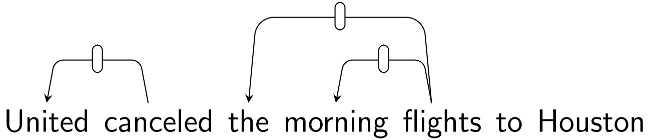


Features - Example configuration:

`United canceled the morning flights to Houston`

Stack: [root, canceled, flights] Buffer: [to, Houston]
R: [canceled $\rightarrow$ United, flights $\rightarrow$ morning, flights $\rightarrow$ the]

Partial structure so far:



What is the next action?
How can we represent this parser state/configuration as features?

Features - Get the basic elements:

(information from stack, buffer or R)
Stack: [root, canceled, flights]   Buffer: [to, Houston]
R: [canceled $\rightarrow$ United, flights $\rightarrow$ morning, flights $\rightarrow$ the]

```
s_w0: flights
s_p0: NOUN

s_w1: canceled
s_p1: VERB

b_w0: to   #buffer

context of top on stack: child1: the, child2: morning
valence of top of stack: 2
```

Features - add features:

Stack: [root, canceled, flights]   Buffer: [to, Houston]
R: [canceled → United, flights → morning, flights → the]

Add features (a unique string = unique feature):

```
# unigram
features.append(('s_w0=flights',1))
features.append(('s_w1=canceled',1))
# feature combinations
features.append(('s_w0=flights,s_p0=NOUN', 1))
features.append(('s_w1=canceled,s_p1=VERB', 1))
# add more!
```

Note1: always add features with value 1!
Note2: in the code you will do this with a format statement, because the feature depends on the current configuration

Outline

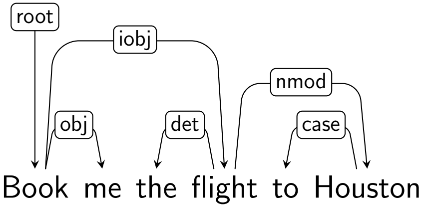
Evaluation

Which proportion of edges is predicted correctly?

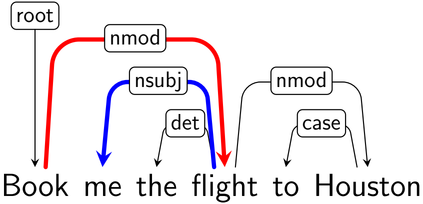
- **Label accuracy** (LA): nodes with correct incoming edge/total number of nodes
- **Unlabeled attachment score** (UAS): nodes with correct parent/total nodes
- **Labeled attachment score** (LAS): nodes with correct parent and edge label / total nodes

Evaluation

gold:

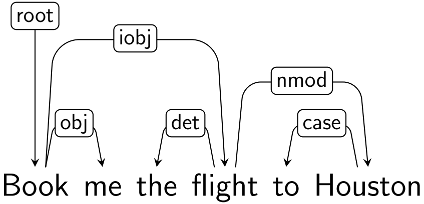


system:

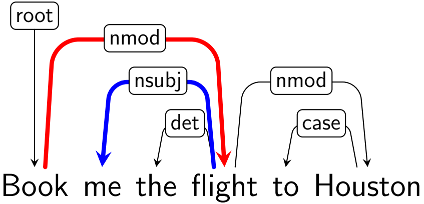


Evaluation

gold:



system:



LAS (labeled): 4/6
UAS (unlabeled): 5/6

Reference

https://web.stanford.edu/~jurafsky/slp3/13.pdf