Language & Technology

Extra: A Glimpse at More Adequate Models

Alëna Aksënova & Aniello De Santo

Stony Brook University
alena.aksenova@stonybrook.edu
aniello.desanto@stonybrook.edu

Current Model

n-**gram models** rule the field.

word n-gram: sequence of n words

character n-gram: sequence of n characters

Example

Sentence	Mary really really likes Marty
Word trigrams (types)	Mary really really really really likes really likes Marty
Char trigrams (types)	Mar, ary, ry_, y_r, _re, rea, eal, all, lly, ly_, y_l, _li, lik, ike, kes, es_, s_M _Ma, art, rty

Applications of n-Grams

- word completion & prediction
- context in spell checking
  good "there are" VS bad "their are"
- possible word detection in spell checking
  does the word contain only possible bigrams of English?
- word choice in OCR
  pick word that maximizes sentence probability

Modus operandi

1. Collect large corpus
2. Extract all bi-/tri/n-gram types and their counts
3. Convert counts to frequency
4. Compute probabilities of relevant alternatives
   e.g. possible word completions
5. Pick choice with highest probability

Example of Probability Maximization

Alternatives

A) John wants to be happy.
B) John wants to he happy.

Bigram probabilities

John wants: 2%
wants to: 5%
to be: 10%
to he: 1%
be happy: 5%
he happy: 1%

$$
\begin{aligned}
P(\text{A}) &= P(\text{John wants}) \times P(\text{wants to}) \times P(\text{to be}) \times P(\text{be happy}) \\
&= 2\% \times 5\% \times 10\% \times 5\% \\
&= 0.005\%
\end{aligned}
$$

$$
\begin{aligned}
P(\text{B}) &= P(\text{John wants}) \times P(\text{wants to}) \times P(\text{to he}) \times P(\text{he happy}) \\
&= 2\% \times 5\% \times 1\% \times 1\% \\
&= \mathbf{0.000001\%}
\end{aligned}
$$

Example of Probability Maximization

Alternatives

A) John wants to be happy.
B) John wants to he happy.

Bigram probabilities

John wants: 2%
wants to: 5%
to be: 10%
to he: 1%
be happy: 5%
he happy: 1%

$$\begin{aligned} P(\text{A}) &= P(\text{John wants}) \times P(\text{wants to}) \times P(\text{to be}) \times P(\text{be happy}) \\ &= 2\% \times 5\% \times 10\% \times 5\% \\ &= 0.005\% \end{aligned}$$

$$\begin{aligned} P(\text{B}) &= P(\text{John wants}) \times P(\text{wants to}) \times P(\text{to he}) \times P(\text{he happy}) \\ &= 2\% \times 5\% \times 1\% \times 1\% \\ &= \mathbf{0.000001\%} \end{aligned}$$

The Limits of Simple Models

n-gram models consider only chunks of sentences:

- no sentence structure
- no meaning
- no discourse information
- no world knowledge

Why There is no Hope for N-Gram Models

n-grams can only consider local information, but
language often uses **non-local information**.

Example

- Suppose the user is typing *May I of* on their phone.
- Do we suggest *off* or *offer* as the best word completion?
- **Bigram Frequencies**

I offer	**0.00014%**
I off	0.00001%

- But what if the user had typed *May I quickly of*?
- **Bigram Frequencies**

quickly offer	0.00000025%
quickly off	**0.000002%**

Scaling Up Doesn't Help

- Okay, so bigrams don't work, but trigrams would:
  I quickly offer is more frequent than *I quickly off*
- But what if the user had typed

May I really quickly of	4-grams!
May I really really quickly of	5-grams!
⋮	

- This isn't feasible, we quickly run into **data sparsity issues**.

Non-Local Dependencies

- Dependencies in language aren't limited to a fixed number n of words, they can span arbitrary distances:
  - *The man that I think Bill thinks I think . . . Bill punched seems angry.*
  - *The men that I think Bill thinks I think . . . Bill punched seem angry.*

The Linguistic Moral of the Story

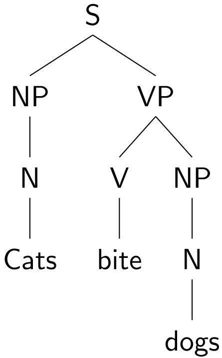
- n-gram models will never work perfectly,
  not even if we had unlimited resources.
- **Reason:** Linguistic dependencies can be unbounded and thus span over more than n words.

How Complex are Sentences?

- We have seen that **n-gram models are insufficient**.
- But what should we use instead?

Sentences Have Hidden Structure

- Linguists have known for a long time that sentences are not just sequences of words.
- They involve a lot of hidden structure ⇒ **trees!**



Some Evidence for Hidden Structure

- Some but not all strings of words can be moved around.

(1) a. It is old ugly dogs that cats bite __.
   b. * It is dogs that cats bite old ugly __.

- Some but not all strings can be coordinated.

(2) a. Cats bite old ugly dogs and scratch young cute dogs.
   b. * Cats bite old ugly and scratch young cute dogs.

- Verbal agreement is not determined by closest noun.

(3) a. The woman is tired.
   b. The women are tired.
   c. * The women that buried the woman is tired.
   d. The women that buried the woman are tired.

Even More Evidence for Hidden Structure

- Questions front the **structurally highest auxiliary**, not the first one in the string.

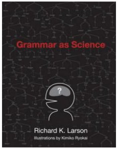
(4) a. The man who is looking for a job is exasperated.

b. * Is the man who __ looking for a job is exasperated?

c. Is the man who is looking for a job __ exasperated?

- lots of evidence from experiments
  self-paced reading, eye tracking, ERP, fMRI

- Lin 311 *Syntax*
- Richard Larson's *Grammar as Science*
- my lecture notes (added to Blackboard)

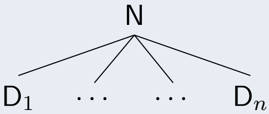


Tree Bigrams

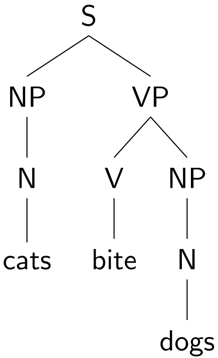
- If sentences are tree structures, then a natural language is not a collection of strings but a collection of trees.
- But how can we describe these trees?

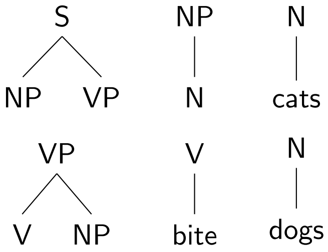
Tree Bigrams

A **tree bigram** consists of a node and one or more daughters:



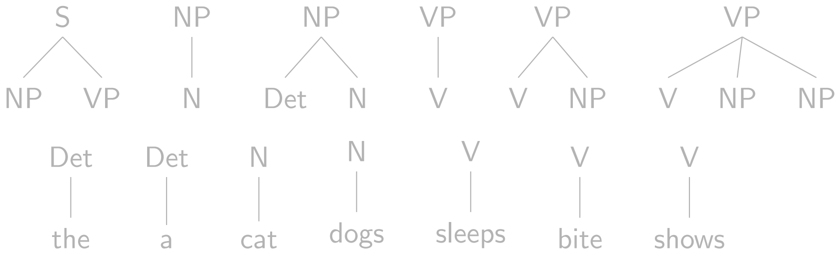
Example: Tree Bigrams in *cats bite dogs*





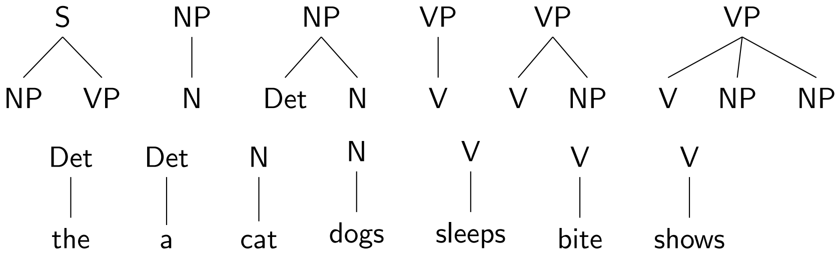
Tree Bigram Grammars

- A **tree bigram grammar** is a collection of well-formed tree bigrams.
- Sentences are constructed by "clicking" tree bigrams together like Lego bricks.



Tree Bigram Grammars

- A **tree bigram grammar** is a collection of well-formed tree bigrams.
- Sentences are constructed by “clicking” tree bigrams together like Lego bricks.



Working with Tree Bigram Grammars: Verifying Trees

It is very easy to verify whether a given tree is licensed by a given grammar.

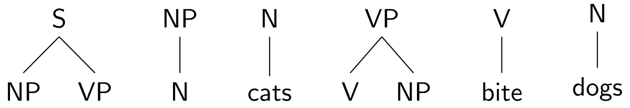
Algorithm: Determining Tree Well-Formedness

Input: grammar G and tree t
Output: True if t is well-formed, False otherwise

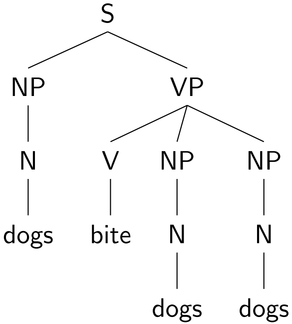
1. extract all tree bigrams of t
2. if at least one tree bigram is not in G, return False
3. otherwise, return True

Example: An Illicit Tree

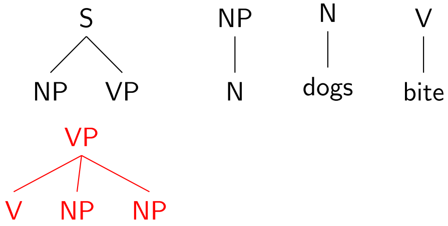
Tree Bigram Grammar:



Input Tree:



Extracted Tree Bigrams:



Adding Probabilities

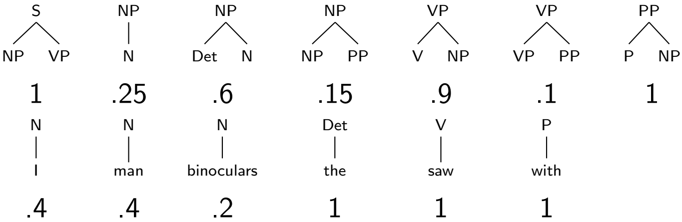
- With bigrams it was often advantageous to add probabilities. This can also be done for tree bigrams.
- **Tree banks** are corpora where every sentence has been annotated with a tree structure.
- We can calculate the frequency of tree bigrams in tree banks and use those to determine various probabilities.

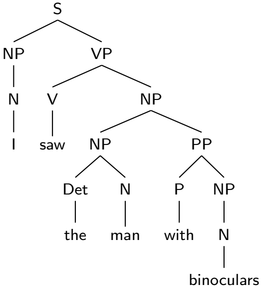
Two Types of Probability

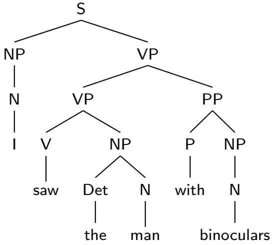
tree probability product of probabilities of all tree bigram tokens

string probability sum of tree probabilities of all trees for the string

Example: Probabilities for an Ambiguous Sentence







Left: $1 \times .25 \times .4 \times .9 \times 1 \times .15 \times .6 \times 1 \times .4 \times 1 \times 1 \times .25 \times .2 = 0.016\%$

Right: $1 \times .25 \times .4 \times .1 \times .9 \times 1 \times .6 \times 1 \times .4 \times 1 \times 1 \times .25 \times .2 = 0.011\%$

Building Trees

- In practice, one hardly ever gets a full tree as input.
- Instead, one usually starts out with the strings of words and has to find the correct tree structure(s).
- The process of inferring tree structure is called **parsing**, and it is **surprisingly difficult**.

Humans are Incredible Parsing Machines

- Humans parse on-the-fly while listening
  $\Rightarrow$ real-time comprehension
- Even our best parsing algorithms take up to n^3 steps, where n is the number of words in the sentence.

# of words	1	2	3	4	5	$\cdots$	10
Computing steps	1	8	27	64	125	$\cdots$	1000

Building Trees

- In practice, one hardly ever gets a full tree as input.
- Instead, one usually starts out with the strings of words and has to find the correct tree structure(s).
- The process of inferring tree structure is called **parsing**, and it is **surprisingly difficult**.

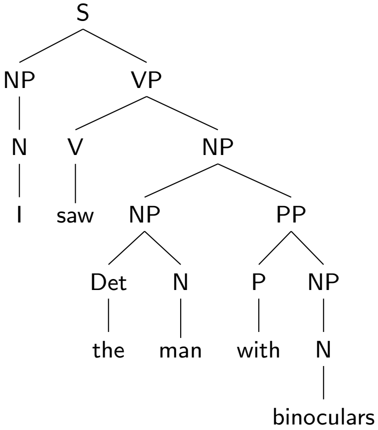
Humans are Incredible Parsing Machines

- Humans parse on-the-fly while listening
  $\Rightarrow$ real-time comprehension
- Even our best parsing algorithms take up to n^3 steps, where n is the number of words in the sentence.

# of words	1	2	3	4	5	$\cdots$	10
Computing steps	1	8	27	64	125	$\cdots$	1000

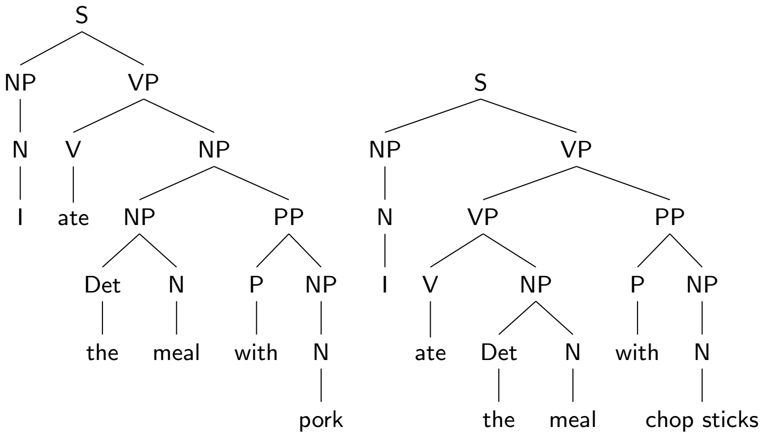
Why Parsing is Harder for Computers

- Sentences can have multiple meanings, they are **ambiguous**.
- This is reflected in **different tree structures**.



Why Parsing is Harder for Computers [cont.]

- But sentences are hardly ever ambiguous in context.



Why Parsing is Harder for Computers [cont.]

- Humans easily handle context and world knowledge.
- That's why humans do not entertain non-sensical tree structures.
- Computers **build all these implausible trees** because they have no grasp of meaning.
- It is this extra work that slows computers down.

Comparison to Parsing of Programming Languages

- Programming languages also have hidden structure that is explicitly specified via a grammar written by the designer of the language.
- This hidden structure is required to translate the program into machine code.
- Programming languages are designed to be free of ambiguity.
- In this case, parsing is **deterministic** and runs at linear speed: parsing a program with n words takes $k \times n + d$ steps.

Moral of the Story

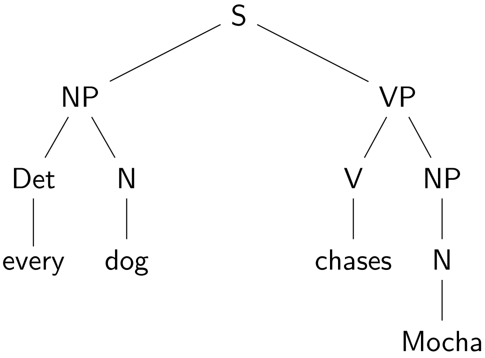
- Without ambiguity, parsing is fast.
- Humans use meaning to keep ambiguity to a minimum.

Trees Make Meaning Easier

Central Idea

- Tree structures encode important relations in sentence
- From the relations we can build the meaning.

Step 1: Annotate tree with grammatical relations

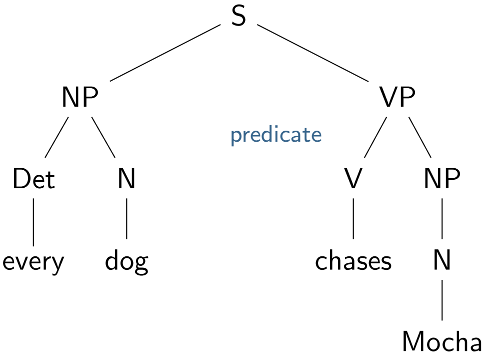


Trees Make Meaning Easier

Central Idea

- Tree structures encode important relations in sentence
- From the relations we can build the meaning.

Step 1: Annotate tree with grammatical relations

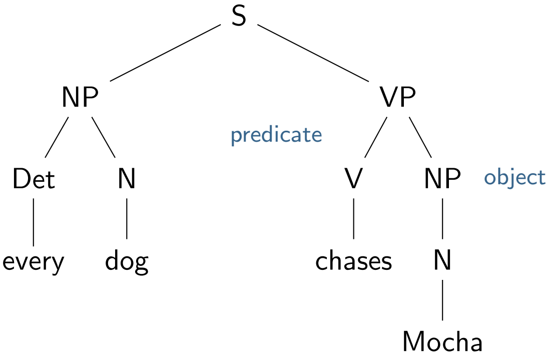


Trees Make Meaning Easier

Central Idea

- Tree structures encode important relations in sentence
- From the relations we can build the meaning.

Step 1: Annotate tree with grammatical relations

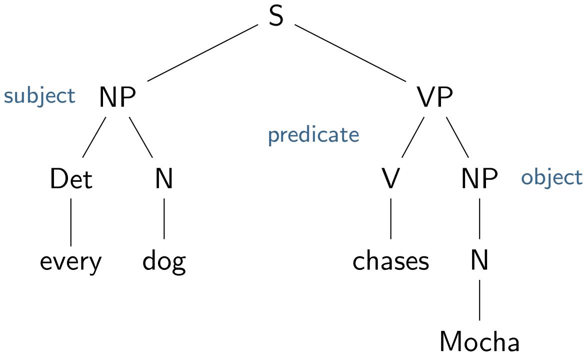


Trees Make Meaning Easier

Central Idea

- Tree structures encode important relations in sentence
- From the relations we can build the meaning.

Step 1: Annotate tree with grammatical relations

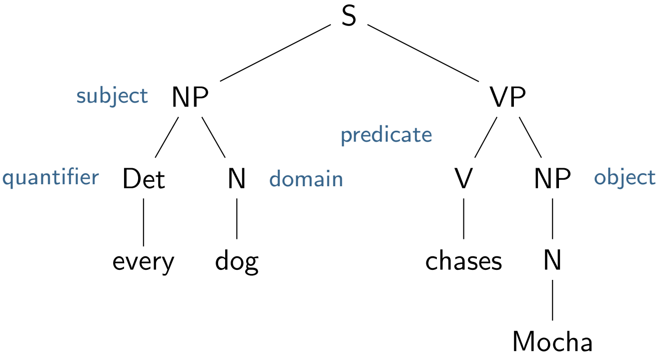


Trees Make Meaning Easier

Central Idea

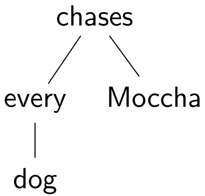
- Tree structures encode important relations in sentence
- From the relations we can build the meaning.

Step 1: Annotate tree with grammatical relations

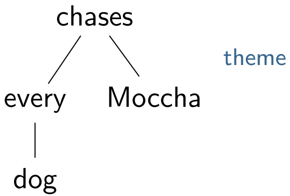


From Phrase Structure Tree to Dependencies

Step 2: Convert annotated tree to dependency tree

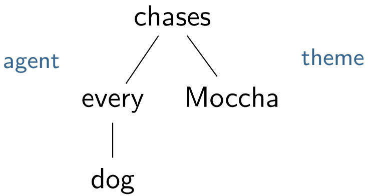


Step 2: Convert annotated tree to dependency tree



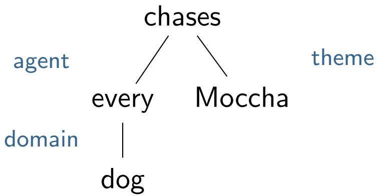
From Phrase Structure Tree to Dependencies

Step 2: Convert annotated tree to dependency tree



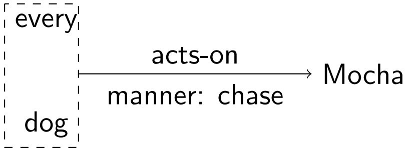
From Phrase Structure Tree to Dependencies

Step 2: Convert annotated tree to dependency tree



From Dependencies to Meaning Representations

Step 3: Convert dependency tree to Abstract Meaning Representation (AMR)

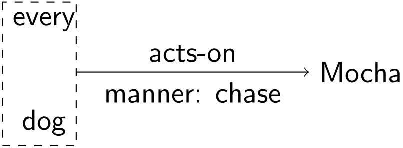


What's the point?

- AMRs encode the abstract meaning dependencies between sentence parts
- This allows us to build a full meaning:
  - Replace individual nodes by their lexical meaning (e.g. vectors)
  - Combine lexical meanings via operators that correspond to dependencies (e.g. dot product)

From Dependencies to Meaning Representations

Step 3: Convert dependency tree to Abstract Meaning Representation (AMR)



What's the point?

- AMRs encode the abstract meaning dependencies between sentence parts
- This allows us to build a full meaning:
  - Replace individual nodes by their lexical meaning (e.g. vectors)
  - Combine lexical meanings via operators that correspond to dependencies (e.g. dot product)