TTIC 31190: Natural Language Processing

Kevin Gimpel

Winter 2016

Lecture 8: Inference in Structured Prediction

Roadmap

- classification
- words
- lexical semantics
- language modeling
- sequence labeling
- syntax and syntactic parsing
- neural network methods in NLP
- semantic compositionality
- semantic parsing
- unsupervised learning
- machine translation and other applications

Roadmap

- classification
- words
- lexical semantics
- language modeling
- sequence labeling
- neural network methods in NLP
- syntax and syntactic parsing
- semantic compositionality
- semantic parsing
- unsupervised learning
- machine translation and other applications

Applications of our Classifier Framework so far

task	input (*x*)	output (*y*)	output space ($\mathcal{L}$)	size of $\mathcal{L}$
text classification	a sentence	gold standard label for *x*	pre-defined, small label set (e.g., {positive, negative})	2-10
word sense disambiguation	instance of a particular word (e.g., *bass* ... its cont...	gold standard ...	pre-defined sense inventory from ...	2-30
learning skip-gram word embeddings	instance ... word in a c...	... a corpus		
part-of-speech tagging	a sentence	gold standard part-of-speech tags for *x*	all possible part-of-speech tag sequences with same length as *x*	$\|P\|^{\|x\|}$

exponential in size of input!

“structured prediction”

Simplest kind of structured prediction: Sequence Labeling

Part-of-Speech Tagging

determiner	verb (past)	prep.	proper noun	proper noun	poss.	adj.	noun
Some	questioned	if	Tim	Cook	’s	first	product

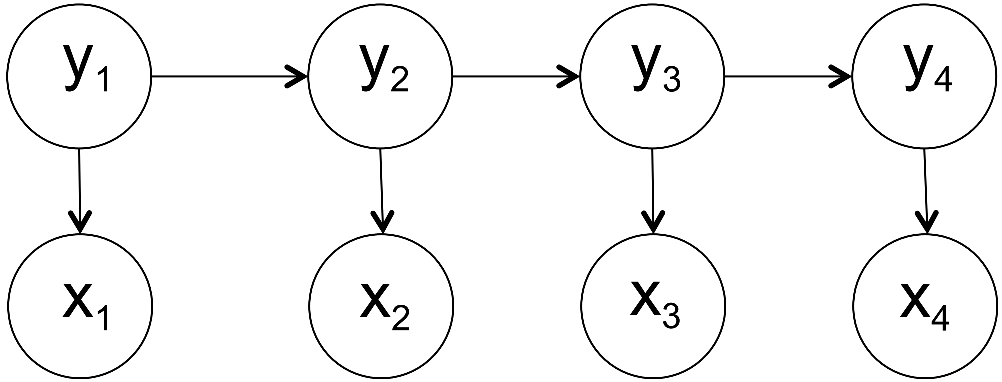
modal	verb	det.	adjective	noun	prep.	proper noun	punc.
would	be	a	breakaway	hit	for	Apple	.

Named Entity Recognition

Some questioned if Tim Cook’s first product would be a breakaway hit for Apple.

PERSON (Tim Cook) **ORGANIZATION** (Apple)

Hidden Markov Models



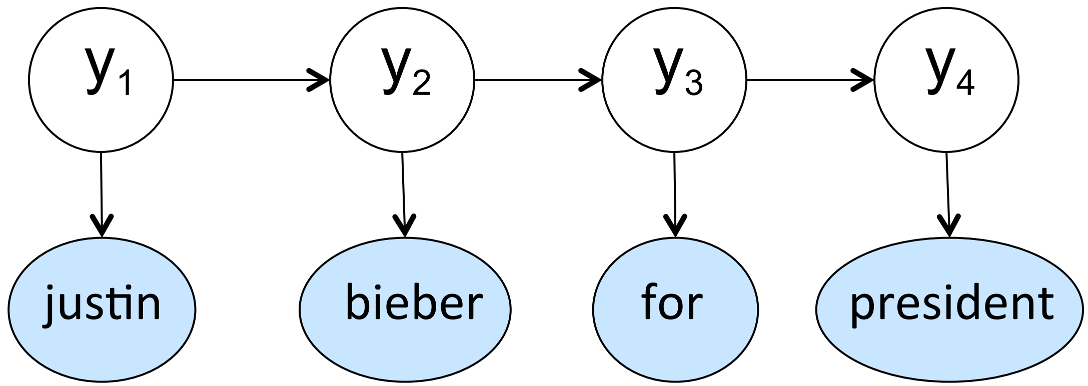
$$p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{y}) = \prod_{i=1}^{|\boldsymbol{x}|} p_{\boldsymbol{\tau}}(y_i \mid y_{i-1}) \; p_{\boldsymbol{\eta}}(x_i \mid y_i)$$

transition parameters: $p_{\boldsymbol{\tau}}(y_i \mid y_{i-1})$

emission parameters: $p_{\boldsymbol{\eta}}(x_i \mid y_i)$

HMMs for Word Clustering

(Brown et al., 1992)



each $y_i \in \mathcal{L}$ is a cluster ID

so, label space is $\mathcal{L} = \{1, 2, ..., 100\}$

simplifying assumption:

each word is in exactly one cluster

HMMs for Word Clustering
(Brown et al., 1992)

- given a set of sentences, how should we learn the parameters of our model?

- how about we use maximum likelihood estimation, e.g.:

$$\underset{\boldsymbol{\theta}}{\operatorname{argmax}} \sum_{i=1}^{N} \log p_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)})$$

- problem: we don't have any $\boldsymbol{y}^{(i)}$'s!

- we only have a set of **unlabeled** sentences: $\{\boldsymbol{x}^{(i)}\}_{i=1}^{N}$

- we want to maximize likelihood, but:
  - our HMM defines $p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{y})$
  - our data only contains $\boldsymbol{x}$
- solution: marginalize out $\boldsymbol{y}$
- this idea underlies most unsupervised learning

$$\operatorname*{argmax}_{\boldsymbol{\theta}} \sum_{i=1}^{N} \log p_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)})$$

$$\Downarrow$$

$$\operatorname*{argmax}_{\boldsymbol{\theta}} \sum_{i=1}^{N} \log \sum_{\boldsymbol{y}} p_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}, \boldsymbol{y})$$

- we want to maximize likelihood, but:
  - our HMM defines $p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{y})$
  - our data only contains $\boldsymbol{x}$
- solution: marginalize out $\boldsymbol{y}$
- this idea underlies most unsupervised learning

a sum over an exponentially-large set

$$\operatorname*{argmax}_{\boldsymbol{\theta}} \sum_{i=1}^{N} \log \sum_{\boldsymbol{y}} p_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}, \boldsymbol{y})$$

- learning requires a sum over an exponentially-large set (of all possible clusterings of the words)

$$\operatorname*{argmax}_{\boldsymbol{\theta}} \sum_{i=1}^{N} \log \sum_{\boldsymbol{y}} p_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}, \boldsymbol{y})$$

- it's actually trivial for Brown clustering (why?)
- for any clustering, we can easily compute the log-likelihood of the data
- problem: there are too many possible clusterings to consider them all!

Algorithm for Brown Clustering

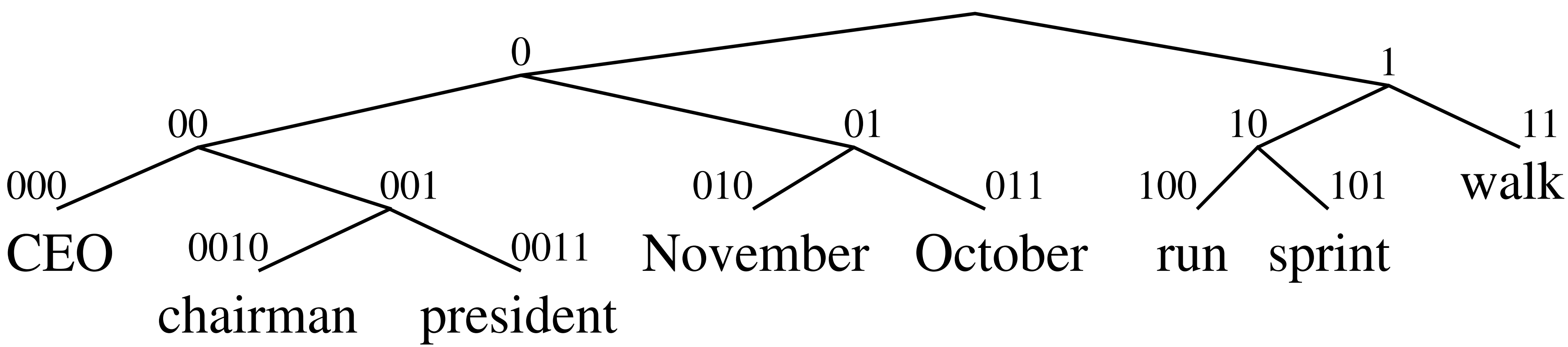
greedy algorithm:

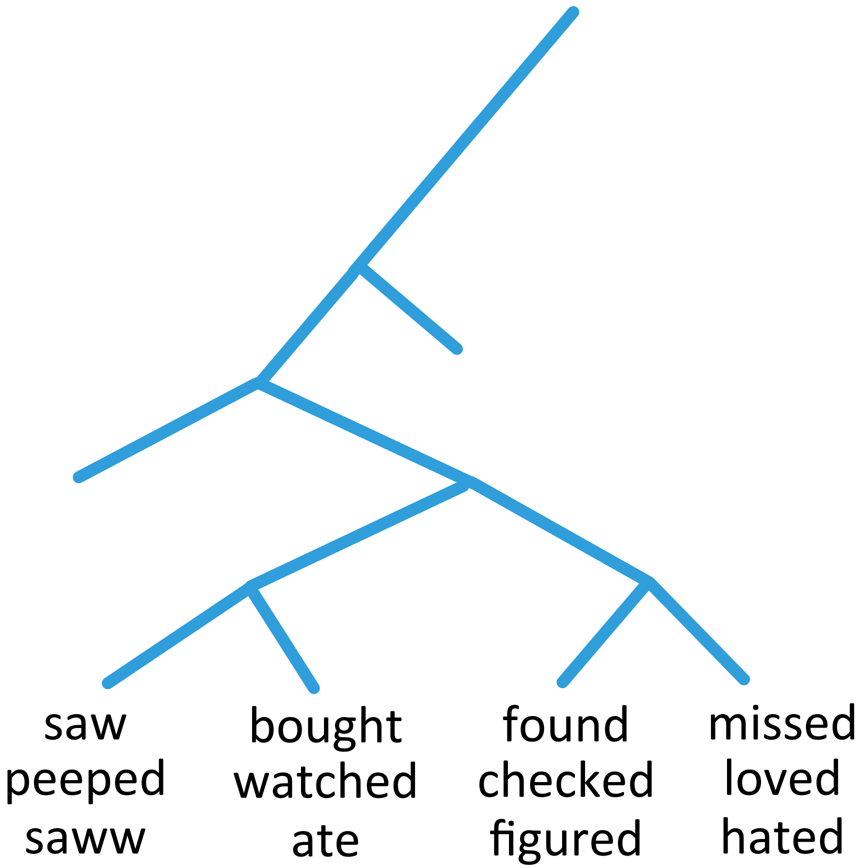
- initialize each word as its own cluster
- greedily merge clusters to improve data likelihood

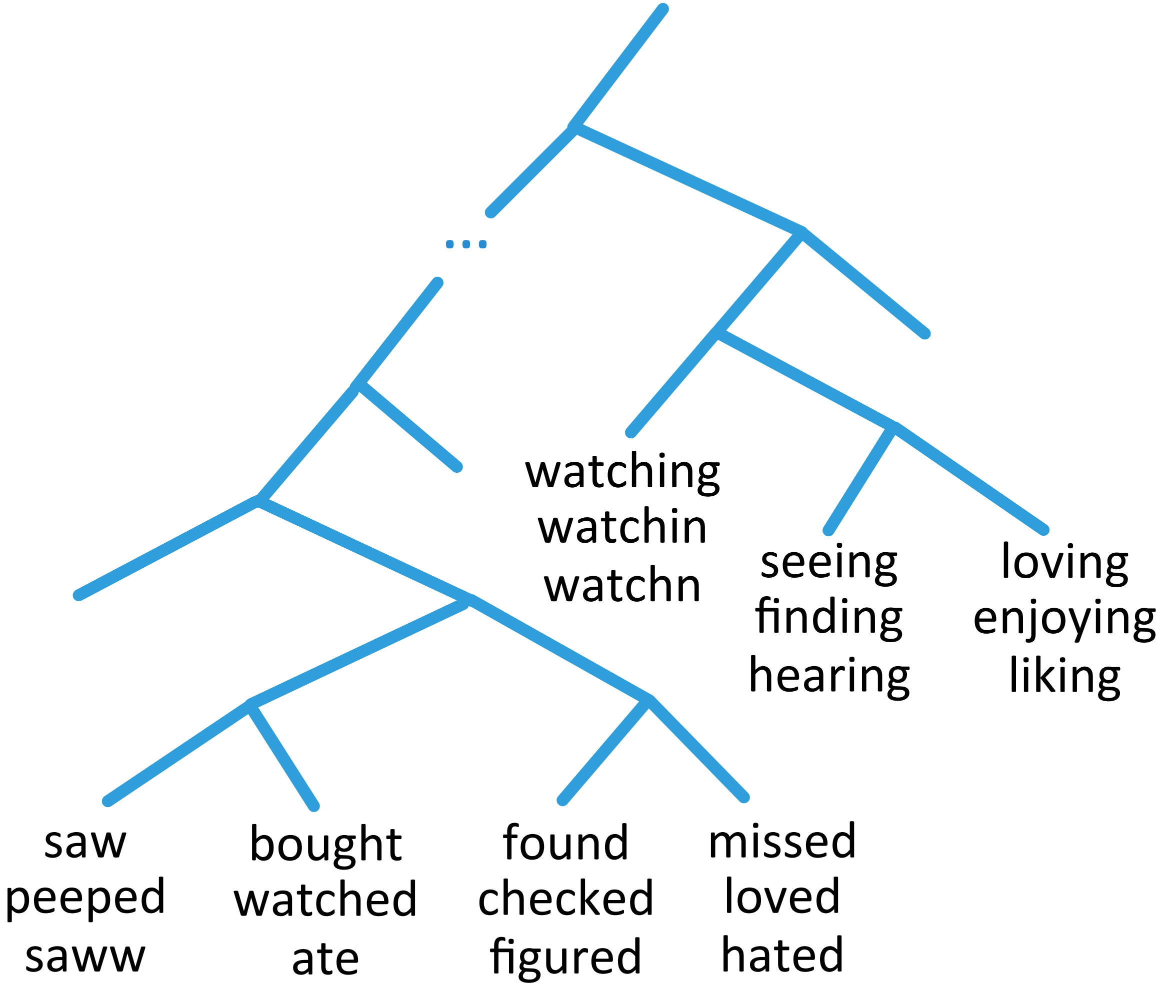
outputs **hierarchical** clustering

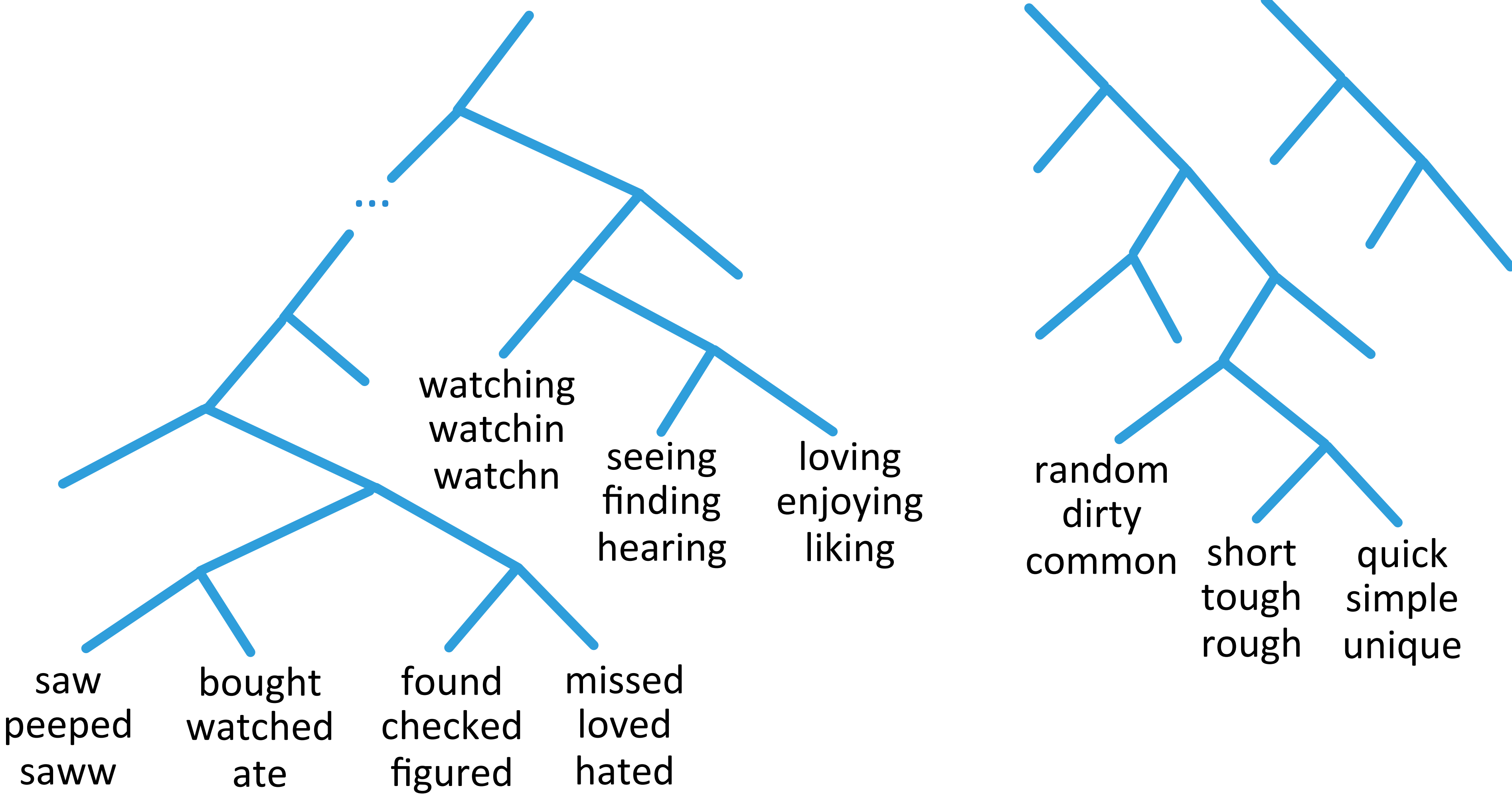
Brown Clusters as vectors

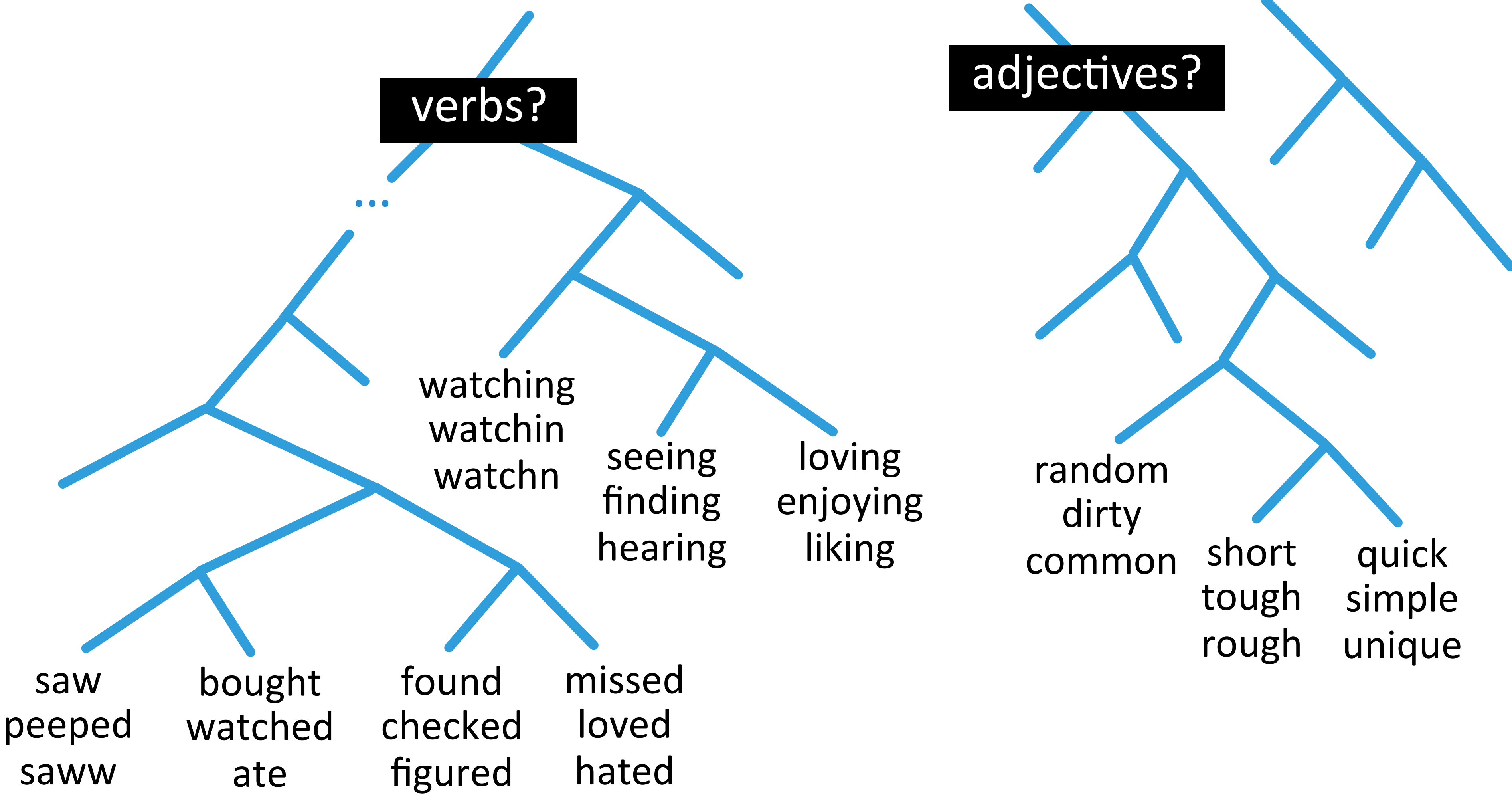
- by tracing order in which clusters are merged, we can build a binary tree from bottom to top
- each word is represented by its binary string = path from root to leaf
- each intermediate node is a cluster
- *chairman* is 0010, "months" = 01, verbs = 1:

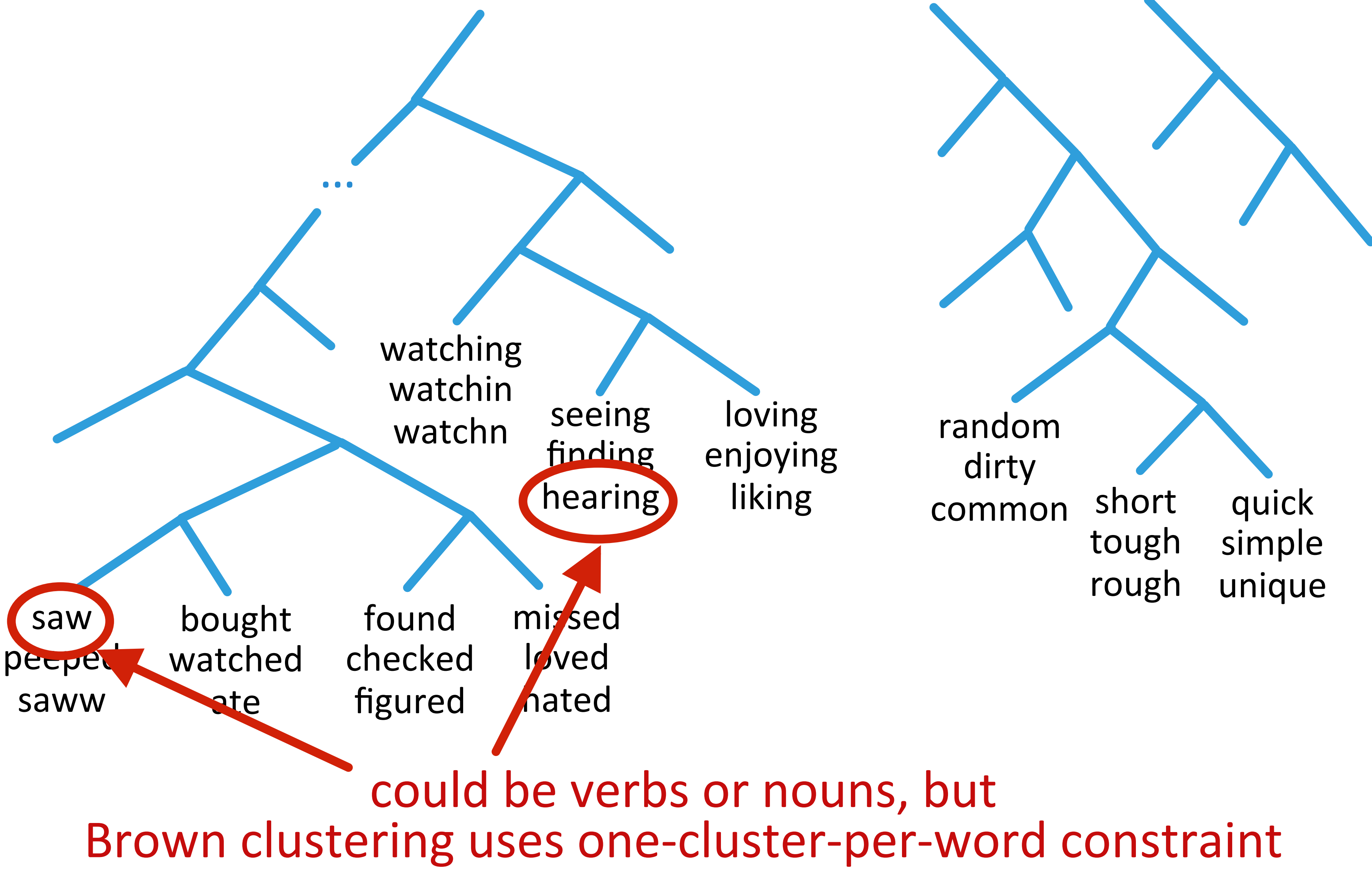



saw
peeped
saww
bought
watched
ate
found
checked
figured
missed
loved
hated

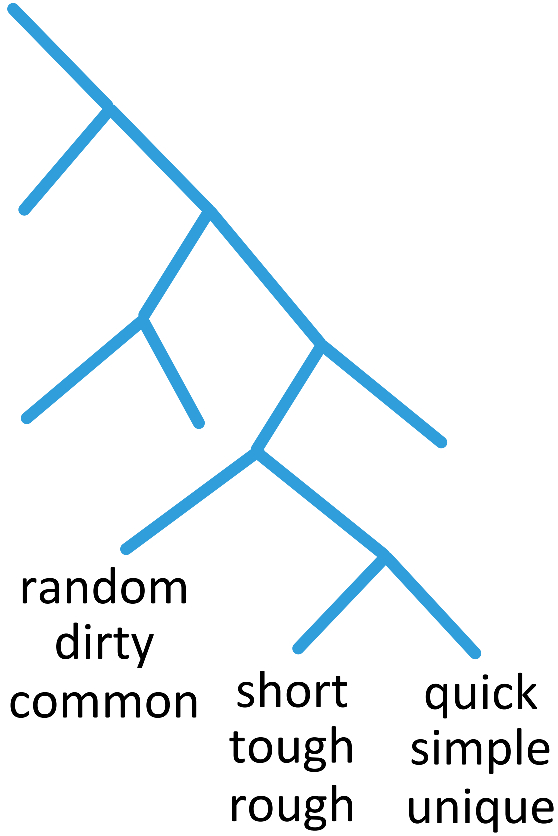

...
watching
watchin
watchn
seeing
finding
hearing
loving
enjoying
liking
saw
peeped
saww
bought
watched
ate
found
checked
figured
missed
loved
hated

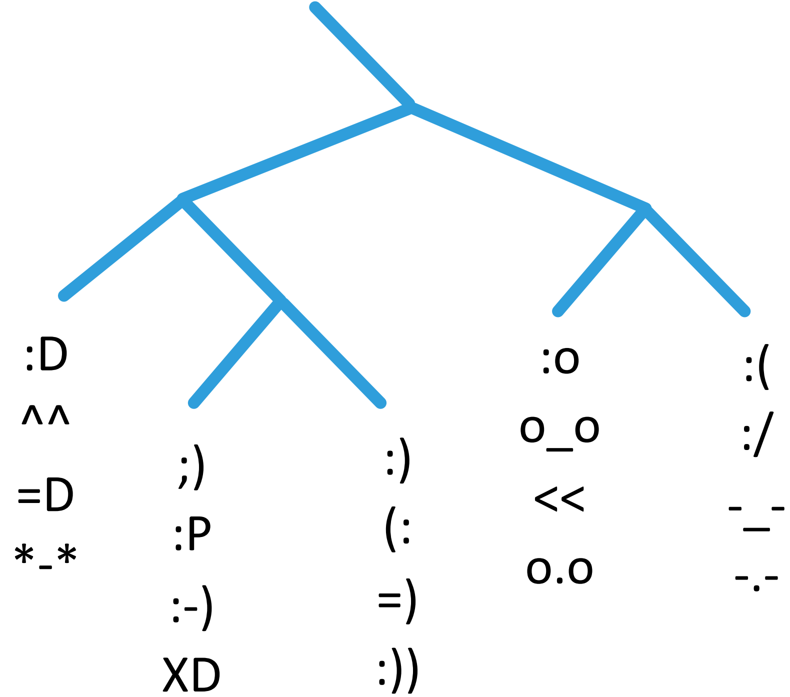

watching
watchin
watchn
seeing
finding
hearing
loving
enjoying
liking
random
dirty
common
short
tough
rough
quick
simple
unique
saw
peeped
saww
bought
watched
ate
found
checked
figured
missed
loved
hated


verbs?
...
watching
watchin
watchn
seeing
finding
hearing
loving
enjoying
liking
saw
peeped
saww
bought
watched
ate
found
checked
figured
missed
loved
hated
adjectives?
random
dirty
common
short
tough
rough
quick
simple
unique



...
watching
watchin
watchn
seeing
finding
hearing
loving
enjoying
liking
random
dirty
common
short
tough
rough
quick
simple
unique
saw
peeped
saww
bought
watched
ate
found
checked
figured
missed
loved
hated
could be verbs or nouns, but
Brown clustering uses one-cluster-per-word constraint


random
dirty
common
short
tough
rough
quick
simple
unique


:D
^^
=D
_
;)
:P
:-)
XD
:)
(:
=)
:))
:o
o_o
<<
o.o
:(
:/
-_-
-.-

$$\underset{\boldsymbol{\theta}}{\operatorname{argmax}} \sum_{i=1}^{N} \log \sum_{\boldsymbol{y}} p_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}, \boldsymbol{y})$$

- though the summation is trivial for Brown clustering, in general we need to be able to compute summations over exponentially-large sets for sequence models and other structured prediction settings

Other Exponentially-Large Problems

inference: solve argmax

$$\text{classify}(x, \boldsymbol{\theta}) = \underset{y}{\text{argmax}} \;\; \text{score}(x, y, \boldsymbol{\theta})$$

- when output is a sequence (or other structure), this argmax requires iterating over an exponentially-large set

Learning requires solving exponentially-hard problems too!

loss	entry *j* of (sub)gradient of loss for linear model
perceptron	$-f_j(\boldsymbol{x}, y) + f_j(\boldsymbol{x}, \hat{y}),\ \text{where } \hat{y} = \text{classify}(\boldsymbol{x}, \boldsymbol{\theta})$
hinge	$-f_j(\boldsymbol{x}, y) + f_j(\boldsymbol{x}, \tilde{y}),\ \text{where } \tilde{y} = \text{costClassify}(\boldsymbol{x}, y, \boldsymbol{\theta})$
log	$-f_j(\boldsymbol{x}, y) + \mathbb{E}_{p_{\boldsymbol{\theta}}(\cdot \mid \boldsymbol{x})}[f_j(\boldsymbol{x}, \cdot)]$

computing each of these terms requires iterating through every possible output

Inference in Structured Prediction

- think of inference as "iterating over the output space"
- specific inference problems:
  - computing argmax in classify() for classification of test data
  - computing argmax in classify() or costClassify() for minimizing perceptron/hinge losses
  - computing feature expectations when minimizing log loss (requires summing over outputs)

- when output space is exponentially-large (e.g., in structured prediction), we need to be clever about how we do this

- today, we'll discuss **dynamic programming** for inference

Dynamic Programming (DP)

- what is dynamic programming?
  - a family of algorithms that break problems into smaller pieces and reuse solutions for those pieces
  - only applicable when the problem has certain properties (**optimal substructure** and **overlapping sub-problems**)

- in this class, we use DP to iterate over exponentially-large output spaces in polynomial time
- we focus on a particular type of DP algorithm: **memoization**

Implementing DP algorithms

- even if your goal is to compute a sum or a max, focus first on **counting mode** (count the number of unique outputs for an input)
- memoization = recursion + saving/reusing solutions
  - start by defining recursive equations
  - "memoize" by creating a table to store all intermediate results from recursive equations, use them when requested

Implementing DP algorithms

- even though we start with counting mode, we need to keep in mind how the model's score function decomposes across parts of the outputs
  - i.e., how "large" are the features? how many items in the output sequence are needed to compute each feature?

Lab

- we will now talk about dynamic programming on the whiteboard and implement some algorithms

Guide to designing/implementing DP algorithms

1. **write down** (or **draw**) all possible outputs for some small input sizes
2. **identify** subproblems that can be solved independently of the overall problem (and confirm that solutions can be reused)
3. **write down** recursive formulas on paper for counting the number of outputs given an input size
4. **work out** (by hand) solutions to your formulas for small inputs, **confirm** that counts match your drawings from step 1
5. **implement** recursive formulas, **confirm** results match drawings, **compute** counts for larger input sizes
6. **implement** memoization in your program: create a table T (e.g., a multi-dimensional array) indexed by signatures of subproblems, save subproblem solutions after computing them, use them when possible
7. **confirm** that solutions computed by memoization match those computed by step 5 for larger input sizes (should be much faster to compute!)
8. finally, change the algorithm from counting to computing sum/max

Counting Sequences

- # of binary sequences of length N
- # of binary sequences of length N with no “00”
- # of binary sequences of length N with no “010”

- # of binary sequences of length N that contain **at least** 3 ones (not necessarily consecutive)

- implement max over binary sequences using backpointers (use a feature that counts instances of “01” and give it a weight of 1)

- extend any of the above to sequences of any alphabet (not just binary)