

CS447: Natural Language Processing

<http://courses.engr.illinois.edu/cs447>

# Lecture 12: Midterm Review

Julia Hockenmaier

*juliahmr@illinois.edu*

3324 Siebel Center

# Topics

- What is NLP and why is NLP hard?
- Finite-State Methods and Morphology
- Language Models
- Classification for NLP
- Neural Nets for NLP
- Vector Semantics and Word Embeddings
- POS Tagging and Sequence Labeling

# Midterm Exam

**When:** Friday, October 11, 2019 in class

**Where:** DCL 1310 (this room)

**What:** Closed book exam:

- You are not allowed to use any cheat sheets, computers, calculators, phones etc.  
(you shouldn't have to anyway)
- Only the material covered in lectures
- Bring a pen (black/blue) or pencil
- **Short questions — we expect short answers!**
- **Tip:** If you can't answer a question, move on to the next one.  
You may not be able to complete the whole exam in the time given — there will be a lot of questions, so first do the ones you know how to answer!

# Question types

## ***Define X:***

Provide a mathematical/formal definition of X

## ***Explain X; Explain what X is/does:***

Use plain English to define X and say what X is/does

## ***Compute X:***

Return X; Show the steps required to calculate it

## ***Draw X:***

Draw a figure of X

## ***Show/Prove that X is true/is the case/...:***

This may require a (typically very simple) proof.

## ***Discuss/Argue whether ...***

Use your knowledge (of X, Y, Z) to argue your point

# Basics: What is NLP and why is it hard?

# What is NLP and why is it hard?

Describe the NLP pipeline.

Explain why ambiguity is one of the core challenges of NLP. Give examples.

Explain the challenges that Zipf's Law poses for NLP.

Describe two different ways for how to represent words in an NLP system. Discuss their relative advantages and disadvantages.

# *“I made her duck”*

What does this sentence mean?

*“duck”*: noun or verb?

*“make”*: “cook X” or “cause X to do Y” ?

*“her”*: “for her” or “belonging to her” ?

Language has different kinds of ambiguity, e.g.:

## **Structural ambiguity**

*“I eat sushi **with tuna**”* vs. *“I eat sushi **with chopsticks**”*

*“I saw the man **with the telescope on the hill**”*

## **Lexical (word sense) ambiguity**

*“I went to the **bank**”*: financial institution or river bank?

## **Referential ambiguity**

*“**John** saw **Jim**. **He** was drinking coffee.”*

# Disambiguation requires statistical models

**Ambiguity** is a core problem for any NLP task

**Statistical models\*** are one of the main tools to deal with ambiguity.

\*more generally: a lot of the models (classifiers, structured prediction models) you learn about in CS446 (Machine Learning) can be used for this purpose. You can learn more about the connection to machine learning in CS546 (Machine learning in Natural Language).

These models need to be trained (estimated, learned) before they can be used (tested).

We will see lots of examples in this class (CS446 is NOT a prerequisite for CS447)



*“I made her duck cassoulet”*

(Cassoulet = a French bean casserole)

The second major problem in NLP is **coverage**:  
We will always encounter unfamiliar words  
and constructions.

Our models need to be able to deal with this.

This means that our models need to be able  
to *generalize* from what they have been trained on  
to what they will be used on.

# Summary: The NLP Pipeline

An NLP system may use some or all of the following steps:

## Tokenizer/Segmenter

to identify words and sentences

## Morphological analyzer/POS-tagger

to identify the part of speech and structure of words

## Word sense disambiguation

to identify the meaning of words

## Syntactic/semantic Parser

to obtain the structure and meaning of sentences

## Coreference resolution/discourse model

to keep track of the various entities and events mentioned

# NLP Pipeline: Assumptions

Each step in the NLP pipeline embellishes the input with **explicit information** about its linguistic structure

POS tagging: parts of speech of word,

Syntactic parsing: grammatical structure of sentence,....

Each step in the NLP pipeline requires **its own explicit (“symbolic”) output representation:**

POS tagging requires a POS tag set

(e.g. NN=common noun singular, NNS = common noun plural, ...)

Syntactic parsing requires constituent or dependency labels

(e.g. NP = noun phrase, or nsubj = nominal subject)

These representations should capture **linguistically appropriate generalizations/abstractions**

Designing these representations requires linguistic expertise

# NLP Pipeline: Shortcomings

Each step in the pipeline relies on a **learned model** that will return the *most likely* representations

- This requires a lot of **annotated training data** for each step
- Annotation is **expensive** and sometimes **difficult** (people are not 100% accurate)
- These models are **never 100% accurate**
- Models make more mistakes if their input contains mistakes

How do we know that we have **captured the “right” generalizations** when designing representations?

- Some representations are **easier to predict** than others
- Some representations are **more useful** for the next steps in the pipeline than others
- But we won't know how easy/useful a representation is until we have a model that we can plug into a particular pipeline

# How many words are there?

How large is the vocabulary of English (or any other language)?

Vocabulary size = nr of distinct word types

Google N-gram corpus: 1 trillion tokens,  
13 million word types that appear 40+ times

If you count words in text, you will find that...

...a few words (mostly closed-class) are very frequent (the, be, to, of, and, a, in, that,...)

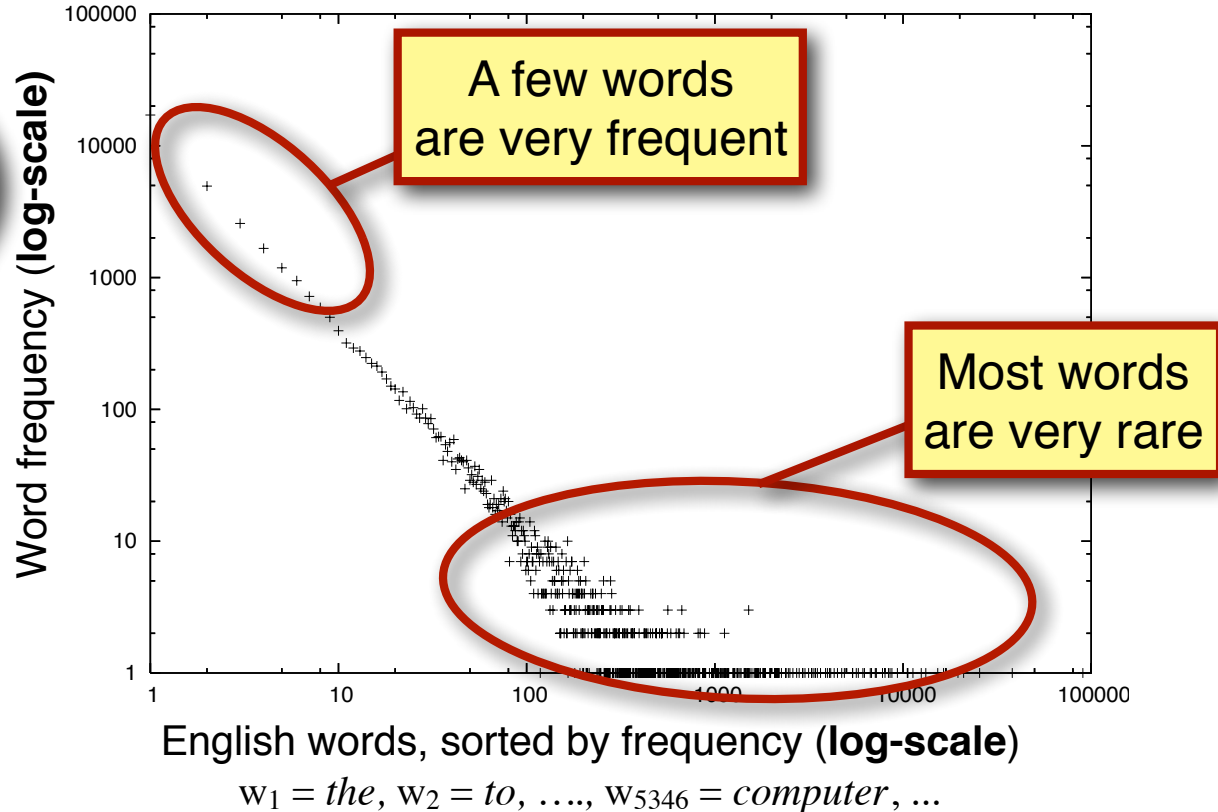
... most words (all open class) are very rare.

... even if you've read a lot of text, you will keep finding words you haven't seen before.

# Zipf's law: the long tail

How many words occur once, twice, 100 times, 1000 times?

the  $r$ -th most common word  $w_r$  has  $P(w_r) \propto 1/r$



In natural language:

- A small number of events (e.g. words) occur with high frequency
- A large number of events occur with very low frequency

# Implications of Zipf's Law for NLP

## The good:

Any text will contain a number of words that are very **common**. We have seen these words often enough that we know (almost) everything about them. These words will help us get at the structure (and possibly meaning) of this text.

## The bad:

Any text will contain a number of words that are **rare**. We know something about these words, but haven't seen them often enough to know everything about them. They may occur with a meaning or a part of speech we haven't seen before.

## The ugly:

Any text will contain a number of words that are **unknown** to us. We have *never* seen them before, but we still need to get at the structure (and meaning) of these texts.

# Dealing with the bad and the ugly

Our systems need to be able to **generalize** from what they have seen to unseen events.

There are two (complementary) approaches to generalization:

- **Linguistics** provides us with insights about the rules and structures in language that we can exploit in the (symbolic) representations we use

E.g.: a finite set of grammar rules is enough to describe an infinite language

- **Machine Learning/Statistics** allows us to learn models (and/or representations) from real data that often work well empirically on unseen data

E.g. most statistical or neural NLP



# How do we represent words?

## Option 1: Words are **atomic symbols**

Can't capture syntactic/semantic relations between words

- Each (surface) word form is its own symbol
- Map different forms of a word to the same symbol
  - **Lemmatization**: map each word to its lemma  
(esp. in English, the lemma is still a word in the language, but lemmatized text is no longer grammatical)
  - **Stemming**: remove endings that differ among word forms  
(no guarantee that the resulting symbol is an actual word)
  - **Normalization**: map all variants of the same word (form) to the same canonical variant (e.g. lowercase everything, normalize spellings, perhaps spell-check)

# How do we represent words?

Option 2: Represent the **structure** of each word

“books” => “book N pl” (or “book V 3rd sg”)

This requires a **morphological analyzer** (more later today)

The output is often a lemma plus morphological information

This is particularly useful for highly inflected languages  
(less so for English or Chinese)

# How do we represent unknown words?

Systems that use machine learning may need to have a unique representation of each word.

## Option 1: the **UNK** token

Replace all rare words (in your training data) with an UNK token (for Unknown word).

Replace *all* unknown words that you come across after training (including rare training words) with the same UNK token

## Option 2: **substring-based** representations

Represent (rare and unknown) words as sequences of characters or substrings

- Byte Pair Encoding: learn which character sequences are common in the vocabulary of your language

# Finite-State Methods and Morphology

# Finite-State Methods and Morphology

What is inflectional morphology? Give examples.

Explain how finite-state transducers can be used for morphological analysis.

Give an example of a language that cannot be recognized by a finite-state automaton.

# Inflectional morphology in English

## Verbs:

Infinitive/present tense: walk, go

3rd person singular present tense (s-form): walks, goes

Simple past: walked, went

Past participle (ed-form): walked, gone

Present participle (ing-form): walking, going

## Nouns:

Common nouns inflect for number:

singular (book) vs. plural (books)

Personal pronouns inflect for person, number, gender, case:

I saw him; he saw me; you saw her; we saw them; they saw us.

# Derivational morphology in English

## Nominalization:

V + -ation: computerization

V+ -er: killer

Adj + -ness: fuzziness

## Negation:

un-: undo, unseen, ...

mis-: mistake,...

## Adjectivization:

V+ -able: doable

N + -al: nationalal

# Morphemes: stems, affixes

**dis-grace-ful-ly**  
**prefix-stem-suffix-suffix**

Many word forms consist of a **stem** plus a number of **affixes** (*prefixes or suffixes*)

Exceptions: *Infixes* are inserted inside the stem

*Circumfixes* (German *gesehen*) surround the stem

**Morphemes**: the smallest (meaningful/grammatical) parts of words.

*Stems* (grace) are often **free morphemes**.

Free morphemes can occur by themselves as words.

*Affixes* (dis-, -ful, -ly) are usually **bound morphemes**.

Bound morphemes *have* to combine with others to form words.



# Morphological parsing

	disgracefully			
dis	grace	ful	ly	
<i>prefix</i>	<i>stem</i>	<i>suffix</i>	<i>suffix</i>	
<b>NEG</b>	grace+N	+ADJ	+ADV	

# Morphological generation

We cannot enumerate all possible English words, but we would like to capture the rules that define whether a string *could* be an English word or not.

That is, we want **a procedure that can generate (or accept) possible English words...**

grace, graceful, gracefully  
disgrace, disgraceful, disgracefully,  
ungraceful, ungracefully,  
undisgraceful, undisgracefully,...

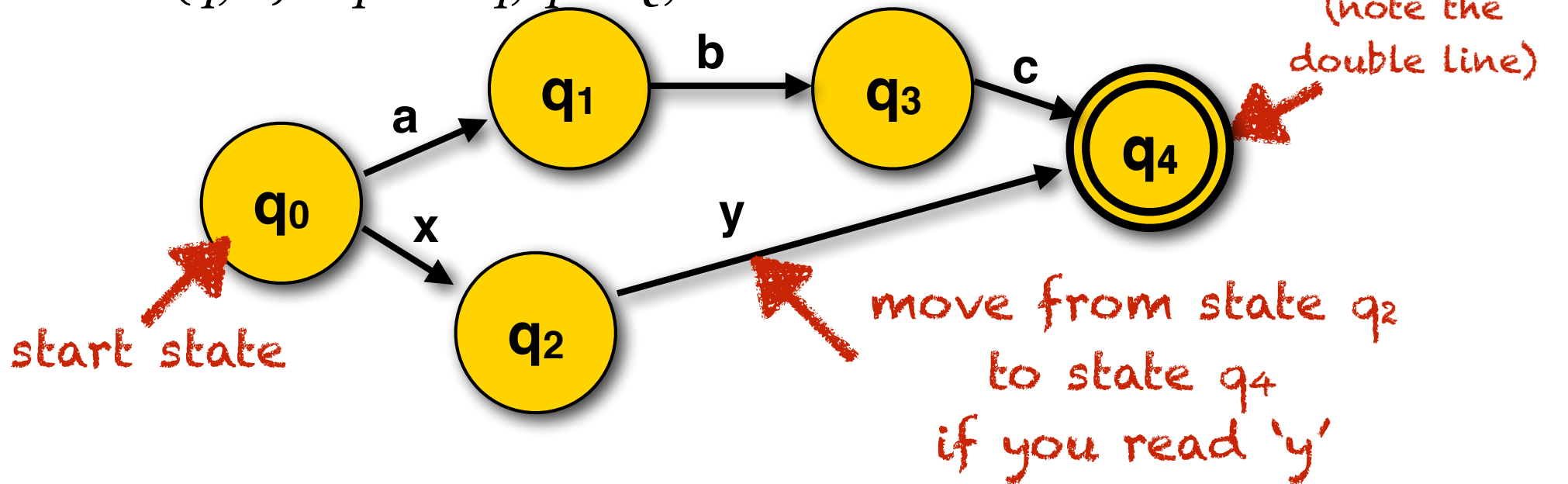
without generating/accepting impossible English words  
\*gracelyful, \*gracefully, \*disungracefully,...

NB: \* is linguists' shorthand for "this is ungrammatical"

# Finite-state automata

A (deterministic) finite-state automaton (FSA) consists of:

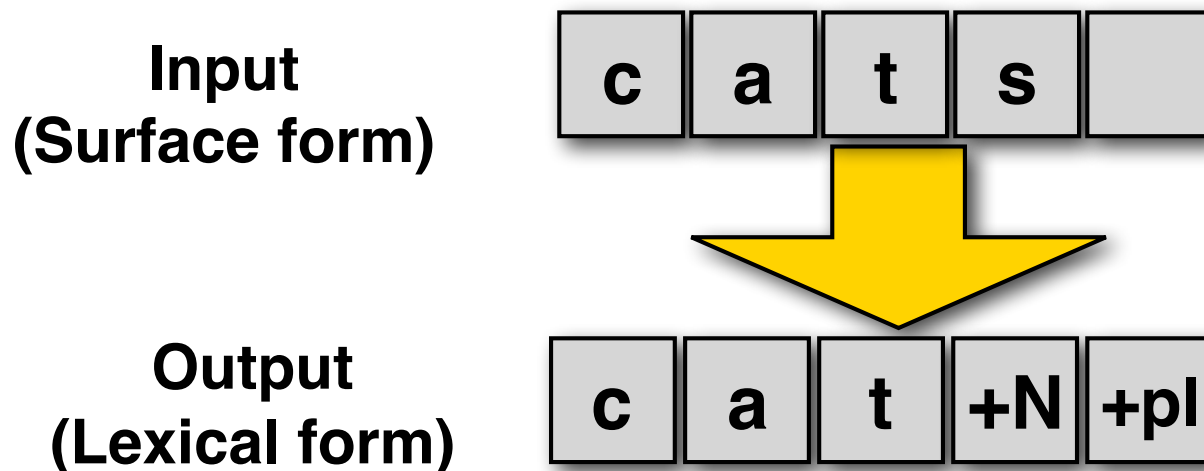
- a **finite set of states**  $Q = \{q_0 \dots q_N\}$ , including a **start state**  $q_0$  and one (or more) **final (=accepting) states** (say,  $q_N$ )
- a (**deterministic**) transition function  $\delta(q, w) = q'$  for  $q, q' \in Q, w \in \Sigma$



# Recognition vs. Analysis

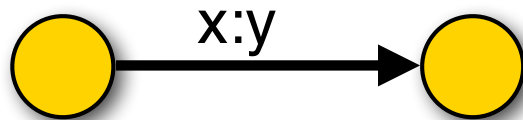
FSAs can recognize (**accept**) a string, but they don't tell us its internal structure.

We need is a machine that maps (**transduces**) the input string into an output string that encodes its structure:

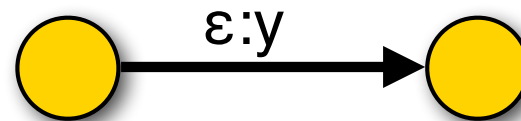


# Finite-state transducers

- FSTs define a **relation** between two regular languages.
- Each state transition maps (**transduces**) a character from the input language to a character (or a sequence of characters) in the output language



- By using the **empty character** ( $\epsilon$ ), characters can be **deleted** ( $x:\epsilon$ ) or **inserted** ( $\epsilon:y$ )




- FSTs can be composed (**cascaded**), allowing us to define **intermediate representations**.

# Finite-state transducers

An FST  $T = L_{in} \times L_{out}$  defines a **relation between two regular languages**  $L_{in}$  and  $L_{out}$ :

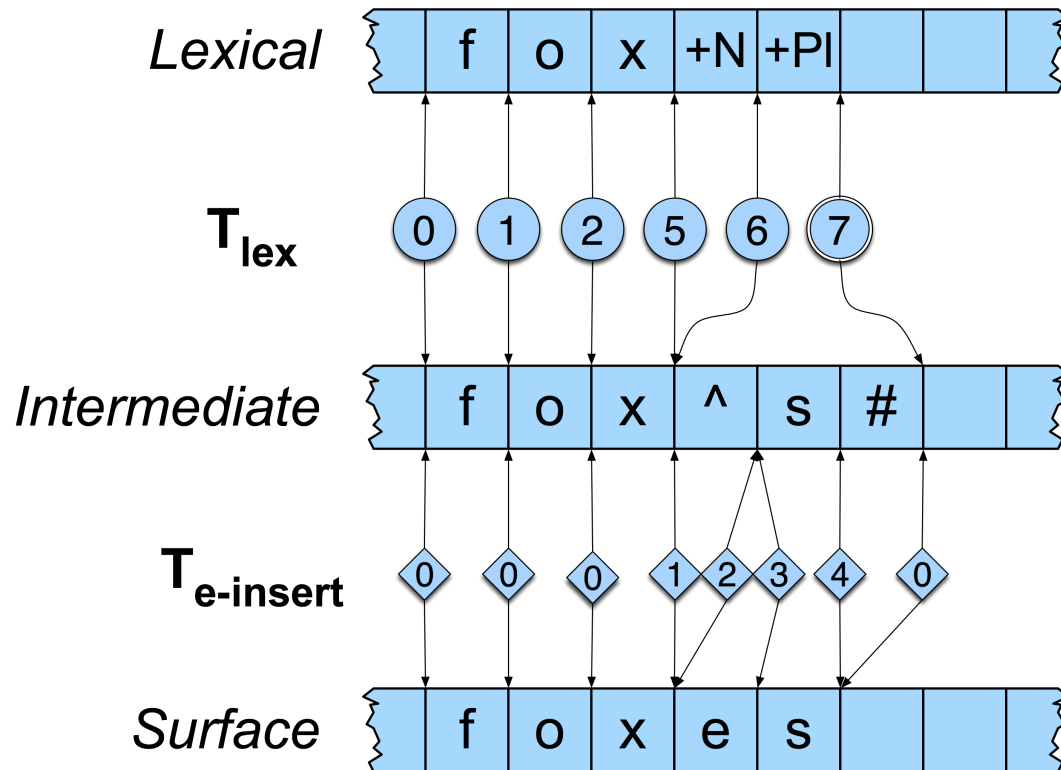
$L_{in} = \{\mathbf{cat}, \mathbf{cats}, \mathbf{fox}, \mathbf{foxes}, \dots\}$

$L_{out} = \{cat+N+sg, cat+N+pl, fox+N+sg, fox+N+pl \dots\}$



$T = \{ \langle \mathbf{cat}, cat+N+sg \rangle, \langle \mathbf{cats}, cat+N+pl \rangle, \langle \mathbf{fox}, fox+N+sg \rangle, \langle \mathbf{foxes}, fox+N+pl \rangle \}$

# FST composition/cascade:



# Language Models



# Language Models

What is a language model?

What independence assumptions does an n-gram language model make?

Describe how to use maximum likelihood estimation for a bigram n-gram model.

Why is it important to use smoothing for language models?

# What is a language model?

Probability distribution over the strings in a language, typically factored into distributions  $P(w_i | \dots)$  for each word:

$$P(\mathbf{w}) = P(w_1 \dots w_n) = \prod_i P(w_i | w_1 \dots w_{i-1})$$

N-gram models assume each word depends only preceding  $n-1$  words:

$$P(w_i | w_1 \dots w_{i-1}) =_{\text{def}} P(w_i | w_{i-n+1} \dots w_{i-1})$$

To handle variable length strings, we assume each string starts with  $n-1$  start-of-sentence symbols (BOS), or  $\langle S \rangle$  and ends in a special end-of-sentence symbol (EOS) or  $\langle \backslash S \rangle$

# Why do we need language models?

Many NLP tasks require **natural language output**:

- **Machine translation**: return text in the target language
- **Speech recognition**: return a transcript of what was spoken
- **Natural language generation**: return natural language text
- **Spell-checking**: return corrected spelling of input

Language models define **probability distributions over (natural language) strings or sentences**.

→ We can use a language model to **score possible output strings** so that we can choose the best (i.e. most likely) one: **if  $P_{LM}(A) > P_{LM}(B)$ , return A, not B**

# Language modeling with N-grams

A language model over a vocabulary  $V$  assigns probabilities to strings drawn from  $V^*$ .

Recall the **chain rule**:

$$P(w^{(1)} \dots w^{(i)}) = P(w^{(1)}) \cdot P(w^{(2)} | w^{(1)}) \cdot \dots \cdot P(w^{(i)} | w^{(i-1)}, \dots, w^{(1)})$$

An **n-gram** language model assumes each word depends only on **the last n-1 words**:

$$P_{ngram}(w^{(1)} \dots w^{(i)}) = P(w^{(1)}) \cdot P(w^{(2)} | w^{(1)}) \cdot \dots \cdot P(w^{(i)} | w^{(i-1)}, \dots, w^{(1-(n+1))})$$

# N-gram models

N-gram models *assume* each word (event) depends only on the previous  $n-1$  words (events):

$$\text{Unigram model: } P(w^{(1)} \dots w^{(N)}) = \prod_{i=1}^N P(w^{(i)})$$

$$\text{Bigram model: } P(w^{(1)} \dots w^{(N)}) = \prod_{i=1}^N P(w^{(i)} | w^{(i-1)})$$

$$\text{Trigram model: } P(w^{(1)} \dots w^{(N)}) = \prod_{i=1}^N P(w^{(i)} | w^{(i-1)}, w^{(i-2)})$$

Such independence assumptions are called **Markov assumptions (of order  $n-1$ )**.

# Learning (estimating) a language model

Where do we get the **parameters of our model** (its actual probabilities) from?

$$P(w^{(i)} = \textit{'the'} \mid w^{(i-1)} = \textit{'on'}) = ???$$

We need (a large amount of) text as **training data** to estimate the parameters of a language model.

The most basic parameter estimation technique:  
**relative frequency estimation** (= counts)

$$P(w^{(i)} = \textit{'the'} \mid w^{(i-1)} = \textit{'on'}) = C(\textit{'on the'}) / C(\textit{'on'})$$

Also called **Maximum Likelihood Estimation (MLE)**

NB: MLE assigns *all* probability mass to events that occur in the training corpus.

# Add-One (Laplace) Smoothing

A really simple way to do smoothing:

**Increment** the actual observed count of every *possible* event (e.g. bigram) by a hallucinated count of 1 (or by a hallucinated count of some  $k$  with  $0 < k < 1$ ).

Shakespeare bigram model (roughly):

0.88 million actual bigram counts  
+ 844.xx million hallucinated bigram counts

Oops. Now almost none of the counts in our model come from actual data. We're back to word salad.

$k$  needs to be really small. But it turns out that that still doesn't work very well.

# How do n-gram models define $P(L)$ ?

An n-gram model defines  $P_{ngram}(w^{(1)} \dots w^{(N)})$  in terms of the **probability of predicting each word**:  $P_{bigram}(w^{(1)} \dots w^{(N)}) = \prod_{i=1 \dots N} P(w^{(i)} | w^{(i-1)})$

With a **fixed vocabulary  $V$** , it's easy to make sure  $P(w^{(i)} | w^{(i-1)})$  is a distribution:  $\sum_{i=1 \dots |V|} P(w_i | w_j) = 1$  and  $\forall_{i,j} 0 \leq P(w_i | w_j) \leq 1$

If  $P(w^{(i)} | w^{(i-1)})$  is a distribution, this model defines **one distribution (over all strings) for each length  $N$**

But the strings of a language  $L$  **don't all have the same length**

English = {“yes!”, “I agree”, “I see you”, ...}

And there is **no  $N_{max}$**  that limits how long strings in  $L$  can get.

**Solution:** the **EOS (end-of-sentence)** token!



# How do n-gram models define $P(L)$ ?

Think of a language model as a **stochastic process**:

- At each time step, randomly pick one more word.
- **Stop** generating more words when the word you pick is a special **end-of-sentence (EOS) token**.

To be able to pick the EOS token, we have to **modify our training data** so that each sentence ends in EOS.

This means our vocabulary is now  $V^{EOS} = V \cup \{EOS\}$

We then get an **actual language model**,  
i.e. a distribution over **strings of any length**

Technically, this is only true because  $P(EOS \mid \dots)$  will be high enough that we are always guaranteed to stop after having generated a finite number of words

***Why do we care about having one model for all lengths?***

We can now compare the probabilities of strings of different lengths, because they're computed by the same distribution.

# Handling unknown words: UNK

## Training:

- Assume a fixed vocabulary (e.g. all words that occur at least  $n$  times in the training corpus)
- Replace all other words in the corpus by a token <UNK>
- Estimate the model on this modified training corpus.

## Testing (e.g to compute probability of a string):

- Replace any words not in the vocabulary by <UNK>

## Refinements:

use different UNK tokens for different types of words (numbers, etc.).

# What about the beginning of the sentence?

In a trigram model

$$P(w^{(1)}w^{(2)}w^{(3)}) = P(w^{(1)})P(w^{(2)} | w^{(1)})P(w^{(3)} | w^{(2)}, w^{(1)})$$

only the third term  $P(w^{(3)} | w^{(2)}, w^{(1)})$  is an actual trigram probability. What about  $P(w^{(1)})$  and  $P(w^{(2)} | w^{(1)})$  ?

**If this bothers you:**

Add  $n-1$  **beginning-of-sentence** (BOS) symbols to each sentence for an  $n$ -gram model:

BOS<sub>1</sub> BOS<sub>2</sub> Alice was ...

Now the unigram and bigram probabilities involve only BOS symbols.

# How do we use language models?

Independently of any application, we can use a language model as a **random sentence generator** (i.e we sample sentences according to their language model probability)

Systems for applications such as machine translation, speech recognition, spell-checking, generation, often produce multiple candidate sentences as output.

- We prefer output sentences  $S_{Out}$  that have a higher probability
- We can use a language model  $P(S_{Out})$  to **score and rank these different candidate output sentences**, e.g. as follows:

$$\operatorname{argmax}_{S_{Out}} P(S_{Out} | \text{Input}) = \operatorname{argmax}_{S_{Out}} P(\text{Input} | S_{Out})P(S_{Out})$$

# Intrinsic vs. Extrinsic Evaluation

Perplexity tells us which LM assigns a higher probability to unseen text

This doesn't necessarily tell us which LM is better for our task (i.e. is better at scoring candidate sentences)

Task-based evaluation:

- Train model A, plug it into your system for performing task T
- Evaluate performance of system A *on task T*.
- Train model B, plug it in, evaluate system B on same task T.
- Compare scores of system A and system B on task T.

# Classification

# Classification

Define multiclass classification.

Explain why it is important to know how well a classifier generalizes to unseen data.

Explain how generative models can be used for classification.

Explain what we mean when we say we use a Bernoulli model in our Naive Bayes text classifier

Explain why accuracy alone may be misleading as an evaluation metric for classification tasks

# Classification tasks

**Classification tasks:** Map **inputs** to a fixed set of **class labels**

**Binary** classification: each input has exactly **one of two classes**

**Multi-class** classification: each input has exactly **one of K classes** ( $K > 2$ )

**Multi-label** classification: each input has **N of K classes** ( $N \geq 1$ , varies per input)

## *What are “inputs”?*

To talk about machine learning mathematically, we often assume **each input item** is represented as a **vector  $\mathbf{x} = (x_1 \dots x_N)$**

(The number of elements  $N$  is fixed, and may be very large)

In NLP, inputs are documents, sentences, words, ....

⇒ How do we represent these as vectors?

Later today we'll assume that each element  $x_i$  in  $(x_1 \dots x_N)$

corresponds to one word type ( $v_i$ ) in the vocabulary  $V = \{v_1, \dots, v_N\}$

— If  $x_i \in \{0, 1\}$ : Does word  $v_i$  occur in the input document?

— If  $x_i \in \{0, 1, 2, \dots\}$ : How often does word  $v_i$  occur in the input document?



# Classification as supervised machine learning

**Classification tasks:** Map inputs to a fixed set of class labels

Underlying assumption: Each input *really* has one (or N) correct labels

Corollary: The **correct mapping** is a function (aka the '**target function**')

How do we **obtain a classifier (model)** for a given task?

- If the target function is very simple (and known), implement it directly
- Otherwise, if we have enough **correctly labeled data**,  
**estimate (aka. learn/train)** a classifier based on that labeled data.

**Supervised machine learning:**

Given (correctly) **labeled training data**, obtain a classifier that predicts these labels as accurately as possible.

Learning is supervised because the learning algorithm can get feedback about how accurate its predictions are from the labels in the training data.

# Probabilistic classifiers

A probabilistic classifier returns the *most likely* class  $y$  for input  $\mathbf{x}$ :

$$y^* = \operatorname{argmax}_y P(Y = y \mid \mathbf{X} = \mathbf{x})$$

**Naive Bayes** uses Bayes Rule:

$$y^* = \operatorname{argmax}_y P(y \mid \mathbf{x}) = \operatorname{argmax}_y P(\mathbf{x} \mid y)P(y)$$

Naive Bayes models the joint distribution:  $P(\mathbf{x} \mid y)P(y) = P(\mathbf{x}, y)$

Joint models are also called **generative** models because we can view them as stochastic processes that *generate* (labeled) items:

Sample/pick a label  $y$  with  $P(y)$ , and then an item  $\mathbf{x}$  with  $P(\mathbf{x} \mid y)$

**Logistic Regression** models  $P(y \mid \mathbf{x})$  directly

This is also called a **discriminative** or **conditional** model, because it only models the probability of the class given the input, and not of the raw data itself.

# Probabilistic classifiers: Naive Bayes

Return the **most likely class  $y$**  for the input  $\mathbf{x}$ :

$$y^* = \mathbf{argmax}_y P(Y = y | \mathbf{X} = \mathbf{x})$$

Naive Bayes classifiers use **Bayes' Rule** (“*the posterior probability  $P(A|B)$  is proportional to prior ( $P(A)$ ) times likelihood  $P(B|A)$* ”)

$$P(A | B) = \frac{P(A, B)}{P(B)} = \frac{P(B | A)P(A)}{P(B)} \propto P(B | A)P(A)$$

$$\begin{aligned} y^* &= \mathbf{argmax}_y P(Y = y | \mathbf{X} = \mathbf{x}) \\ &= \mathbf{argmax}_y \frac{P(\mathbf{X} = \mathbf{x} | Y = y)P(Y = y)}{P(\mathbf{X} = \mathbf{x})} \quad [\text{Bayes' Rule}] \\ &= \mathbf{argmax}_y P(\mathbf{X} = \mathbf{x} | Y = y)P(Y = y) \quad [P(\mathbf{X}) \text{ doesn't change } \mathbf{argmax}_y] \end{aligned}$$

# The Naive Bayes Classifier

Assign class  $y^*$  to input  $\mathbf{x} = (x_1 \dots x_n)$  if

$$y^* = \mathbf{argmax}_y P(Y = y) \prod_{i=1..n} P(X_i = x_i | Y = y)$$

$P(Y = y)$  is the prior class probability (estimated as the fraction of items in the training data with class  $y$ )

$P(X_i = x_i | Y = y)$  is the (class-conditional) likelihood of the feature  $x_i$ .

There are different ways to model this probability

# Modeling $P(\mathbf{X} = \mathbf{x} | Y = y)P(Y = y)$

$P(Y = y)$  is the “prior” class probability

We can estimate this as the fraction of documents in the training data that have class  $y$ :

$$\hat{P}(Y = y) = \frac{\text{\#documents } \langle \mathbf{x}_i, y_i \rangle \in D_{train} \text{ with } y_i = y}{\text{\#documents } \langle \mathbf{x}_i, y_i \rangle \in D_{train}}$$

$P(\mathbf{X} = \mathbf{x} | Y = y)$  is the “likelihood” of the input  $\mathbf{x}$   
 $\mathbf{x} = (x_1 \dots x_n)$  is a vector; each  $x_i \approx$  a word in our vocabulary

Let’s make a (naive) independence assumption:

$$P(\mathbf{X} = \langle x_1, \dots, x_n \rangle | Y = y) := \prod_{i=1..n} P(X_i = x_i | Y = y)$$

Now we need to multiply together all  $P(X_i = x_i | Y = y)$

# $P(X_i = x_i | Y = y)$ as Bernoulli

$P(X_i = x_i | Y = y)$  is a **Bernoulli** distribution ( $x_i \in \{0,1\}$ )

$P(X_i = 1 | Y = y)$  is the probability that **word  $v_i$  occurs** in a document of class  $y$ .

$P(X_i = 0 | Y = y)$  is the probability that **word  $v_i$  does not occur** in a document of class  $y$

Estimation:

$$\hat{P}(X_i = 1 | Y = y) = \frac{\text{\#docs } \langle \mathbf{x}_i, y_i \rangle \in D_{train} \text{ with } y_i = y \text{ in which } x_i \text{ occurs}}{\text{\#docs } \langle \mathbf{x}_i, y_i \rangle \in D_{train} \text{ with } y_i = y}$$

$$\hat{P}(X_i = 0 | Y = y) = \frac{\text{\#docs } \langle \mathbf{x}_i, y_i \rangle \in D_{train} \text{ with } y_i = y \text{ in which } x_i \text{ does not occur}}{\text{\#docs } \langle \mathbf{x}_i, y_i \rangle \in D_{train} \text{ with } y_i = y}$$

# $P(\mathbf{X}_i = \mathbf{x}_i | Y = y)$ as Multinomial

$P(\mathbf{X}_i = \mathbf{x}_i | Y = y)$  is a **Multinomial**: ( $x_i \in \{0, 1, 2, \dots\}$ )

$P(X_i = x_i | Y = y)$  is the probability that word  $v_i$  occurs with frequency  $x_i$  ( $= 0, 1, 2, \dots$ ) in a document of class  $y$ .

We can estimate the **unigram probability**  $P(v_i | Y = y)$  of word  $v_i$  in all documents of class  $y$  as

$$\hat{P}(v_i | Y = y) = \frac{\#v_i \text{ in all docs } \in D_{\text{train}} \text{ of class } y}{\#\text{words in all docs } \in D_{\text{train}} \text{ of class } y}$$

or **with add-one smoothing** (with  $N$  words in vocab  $V$ ):

$$\hat{P}(v_i | Y = y) = \frac{(\#v_i \text{ in all docs } \in D_{\text{train}} \text{ of class } y) + 1}{(\#\text{words in all docs } \in D_{\text{train}} \text{ of class } y) + N}$$

# Unigram probabilities $P(v_i | Y = y)$

We can estimate the **unigram probability**  $P(v_i | Y = y)$  of word  $v_i$  in all documents of class  $y$  as

$$\hat{P}(v_i | Y = y) = \frac{\#v_i \text{ in all docs } \in D_{\text{train}} \text{ of class } y}{\#\text{words in all docs } \in D_{\text{train}} \text{ of class } y}$$

or **with add-one smoothing** (with  $N$  words in vocab  $V$ ):

$$\hat{P}(v_i | Y = y) = \frac{(\#v_i \text{ in all docs } \in D_{\text{train}} \text{ of class } y) + 1}{(\#\text{words in all docs } \in D_{\text{train}} \text{ of class } y) + N}$$



# Evaluating Classifiers

## Evaluation setup:

Split data into separate **training**, (**development**) and **test** sets.



## Better setup: **n-fold cross validation**:

Split data into  $n$  sets of equal size

Run  $n$  experiments, using set  $i$  to test and remainder to train



This gives average, maximal and minimal accuracies

## When **comparing two classifiers**:

Use the **same** test and training data with the same classes

# Evaluation Metrics

**Accuracy:** How many documents in the test data did you classify correctly?

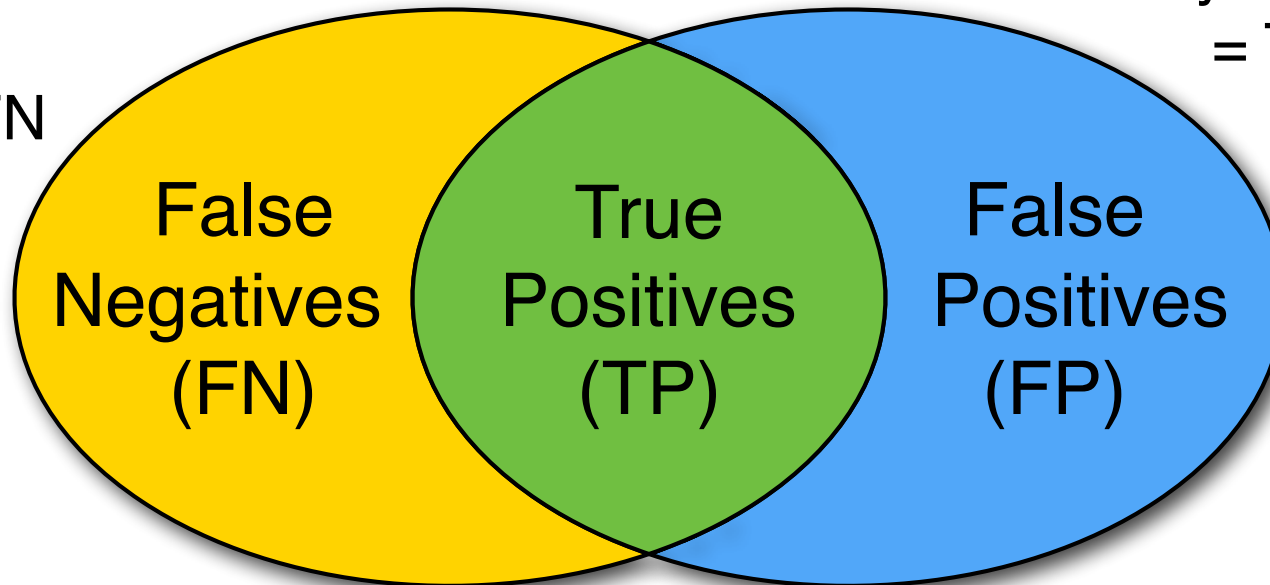
It's easy to get high accuracy if one class is very common (just label everything as that class)

But that would be a pretty useless classifier

# Precision, recall, f-measure

Items labeled X  
in the gold standard  
(‘truth’)  
= TP + FN

Items labeled X  
by the system  
= TP + FP



**Precision:**  $P = \frac{TP}{TP + FP}$

**Recall:**  $R = \frac{TP}{TP + FN}$

**F-measure:** harmonic mean of precision and recall

$$F = \frac{2 \cdot P \cdot R}{P + R}$$

# Confusion matrices

		<i>gold labels</i>					
		urgent	normal	spam			
<i>system output</i>	urgent	8	10	1	<b>precision<sub>u</sub></b> = $\frac{8}{8+10+1}$		
	normal	5	60	50	<b>precision<sub>n</sub></b> = $\frac{60}{5+60+50}$		
	spam	3	30	200	<b>precision<sub>s</sub></b> = $\frac{200}{3+30+200}$		
		<b>recall<sub>u</sub></b> = $\frac{8}{8+5+3}$	<b>recall<sub>n</sub></b> = $\frac{60}{10+60+30}$	<b>recall<sub>s</sub></b> = $\frac{200}{1+50+200}$			

**Figure 4.5** Confusion matrix for a three-class categorization task, showing for each pair of classes  $(c_1, c_2)$ , how many documents from  $c_1$  were (in)correctly assigned to  $c_2$

# Micro-average vs Macro-average

Class 1: Urgent			Class 2: Normal			Class 3: Spam			Pooled		
	true urgent	true not		true normal	true not		true spam	true not		true yes	true no
system urgent	8	11	system normal	60	55	system spam	200	33	system yes	268	99
system not	8	340	system not	40	212	system not	51	83	system no	99	635
precision = $\frac{8}{8+11} = .42$			precision = $\frac{60}{60+55} = .52$			precision = $\frac{200}{200+33} = .86$			microaverage precision = $\frac{268}{268+99} = .73$		
macroaverage precision = $\frac{.42+.52+.86}{3} = .60$											

**Figure 4.6** Separate contingency tables for the 3 classes from the previous figure, showing the pooled contingency table and the microaveraged and macroaveraged precision.

Macro-average: average the precision over all classes  
(regardless of how common each class is)

Micro-average: average the precision over all items  
(regardless of which class they have)

# $P(Y | \mathbf{X})$ with Logistic Regression

**Task:** Model  $P(y | \mathbf{x})$  for any input (feature) vector  $\mathbf{x}=(x_1, \dots, x_n)$

**Idea:** Learn **feature weights**  $\mathbf{w}=(w_1, \dots, w_n)$  (and a bias term  $b$ ) to capture how important each feature  $x_i$  is for predicting the class  $y$

For **binary classification** ( $y \in \{0, 1\}$ ), (standard) logistic regression uses the **sigmoid** function:

$$P(Y=1 | \mathbf{x}) = \sigma(\mathbf{w}\mathbf{x} + b) = \frac{1}{1 + \exp(-(\mathbf{w}\mathbf{x} + b))}$$

Parameters to learn: one **feature weight vector**  $\mathbf{w}$  and one **bias term**  $b$

For **multiclass classification** ( $y \in \{0, 1, \dots, K\}$ ), multinomial logistic regression uses the **softmax** function:

$$P(Y=y_i | \mathbf{x}) = \text{softmax}(\mathbf{z})_i = \frac{\exp(z_i)}{\sum_{j=1}^K \exp(z_j)} = \frac{\exp(-(\mathbf{w}_i\mathbf{x} + b_i))}{\sum_{j=1}^K \exp(-(\mathbf{w}_j\mathbf{x} + b_j))}$$

Parameters to learn: one **feature weight vector**  $\mathbf{w}$  and one **bias term**  $b$  **per class**.

# Using Logistic Regression

How do we create a (binary) logistic regression classifier?

- 1) **Design**: Decide how to map raw inputs to feature vectors  $\mathbf{x}$
- 2) **Training**: Learn parameters  $\mathbf{w}$  and  $b$  on training data
- 3) **Testing**: Use the classifier to classify unseen inputs

**Feature Design**: from raw inputs to feature vectors  $\mathbf{x}$

In a generative model, we have to learn a model for  $P(\mathbf{x} | y)$ .

To guarantee that we get a proper distribution ( $\sum_{\mathbf{x}} P(\mathbf{x} | y) = 1$ ), we have to assume that the features (elements of  $\mathbf{x}$ ) are independent (more precisely, conditionally independent given  $y$ ),

In a conditional model, we only have to learn  $P(y | \mathbf{x})$ , not for  $P(\mathbf{x} | y)$ .

Advantage: Because we don't need a distribution over  $\mathbf{x}$ , we do not need to assume that our features  $x_1, \dots, x_n$  are independent.

# Feature Design: From raw inputs to feature vectors $\mathbf{x}$

## Feature design for **generative models (Naive Bayes)**:

- In a generative model, we have to learn a model for  $P(\mathbf{x} | y)$ .
- Getting a proper distribution ( $\sum_{\mathbf{x}} P(\mathbf{x} | y) = 1$ ) is difficult
- NB assumes that the **features (elements of  $\mathbf{x}$ ) are independent\*** and defines  $P(\mathbf{x} | y) = \prod_i P(x_i | y)$  via a multinomial or Bernoulli (\*more precisely, conditionally independent given  $y$ )
- Different kinds of feature values (boolean, integer, real) require different kinds of distributions  $P(x_i | y)$  (Bernoulli, multinomial, etc.)

## Feature design for **conditional models (Logistic Regression)**:

- In a conditional model, we only have to learn  $P(y | \mathbf{x})$
- It is much easier to get a proper distribution ( $\sum_y P(y | \mathbf{x}) = 1$ )
- **We don't need to assume that our features are independent**
- Any numerical feature  $x_i$  can be used to compute  $\exp(w_j x_i)$



# Useful features that are not independent

Different features can *overlap* in the input

(e.g. we can model both unigrams and bigrams, or overlapping bigrams)

Features can capture *properties* of the input

(e.g. whether words are capitalized, in all-caps, contain particular [classes of] letters or characters, etc.)

This also makes it easy to use predefined dictionaries of words (e.g. for sentiment analysis, or gazetteers for names):

Is this word “positive” (*happy*) or “negative” (*awful*)?

Is this the name of a person (*Smith*) or city (*Boston*) [it may be both (*Paris*)]

Features can capture *combinations* of properties

(e.g. whether a word is capitalized *and* ends in a full stop)

We can use the *outputs of other classifiers* as features

(e.g. to combine weak [less accurate] classifiers for the same task, or to get at complex properties of the input that require a learned classifier)

# Learning = Optimization = Loss Minimization

## Learning = parameter estimation = optimization:

Given a particular class of model (logistic regression, Naive Bayes, ...) and data  $D_{\text{train}}$ , find the **best parameters** for this class of model on  $D_{\text{train}}$

If the model is a probabilistic classifier, think of optimization as **Maximum Likelihood Estimation (MLE)**

*“Best” = return (among all possible parameters for models of this class) parameters that assign the **largest probability** to  $D_{\text{train}}$*

In general (incl. for probabilistic classifiers), think of optimization as **Loss Minimization**:

*“Best” = return (among all possible parameters for models of this class) parameters that have the **smallest loss** on  $D_{\text{train}}$*

**“Loss”**: how bad are the predictions of a model?

*The **loss function** we use to measure loss depends on the class of model*

*$L(\hat{y}, y)$ : how bad is it to predict  $\hat{y}$  if the correct label is  $y$  ?*

# Conditional MLE $\Rightarrow$ Cross-Entropy Loss

Conditional MLE: *Maximize probability of labels* in  $D_{\text{train}}$

$$(\mathbf{w}^*, b^*) = \operatorname{argmax}_{(\mathbf{w}, b)} \prod_{(\mathbf{x}_i, y_i) \in D_{\text{train}}} P(y_i | \mathbf{x}_i)$$

$\Rightarrow$  Maximize  $P(1 | \mathbf{x}_i)$  for any  $(\mathbf{x}_i, 1)$  with a *positive* label in  $D_{\text{train}}$

$\Rightarrow$  Maximize  $P(0 | \mathbf{x}_i)$  for any  $(\mathbf{x}_i, 0)$  with a *negative* label in  $D_{\text{train}}$

Equivalently: *Minimize negative log prob. of labels* in  $D_{\text{train}}$

$P(y_i | \mathbf{x}) = 0 \Leftrightarrow -\log(P(y_i | \mathbf{x})) = +\infty$  if  $y_i$  is the correct label for  $\mathbf{x}$ , this is the worst possible model

$P(y_i | \mathbf{x}) = 1 \Leftrightarrow -\log(P(y_i | \mathbf{x})) = 0$  if  $y_i$  is the correct label for  $\mathbf{x}$ , this is the best possible model

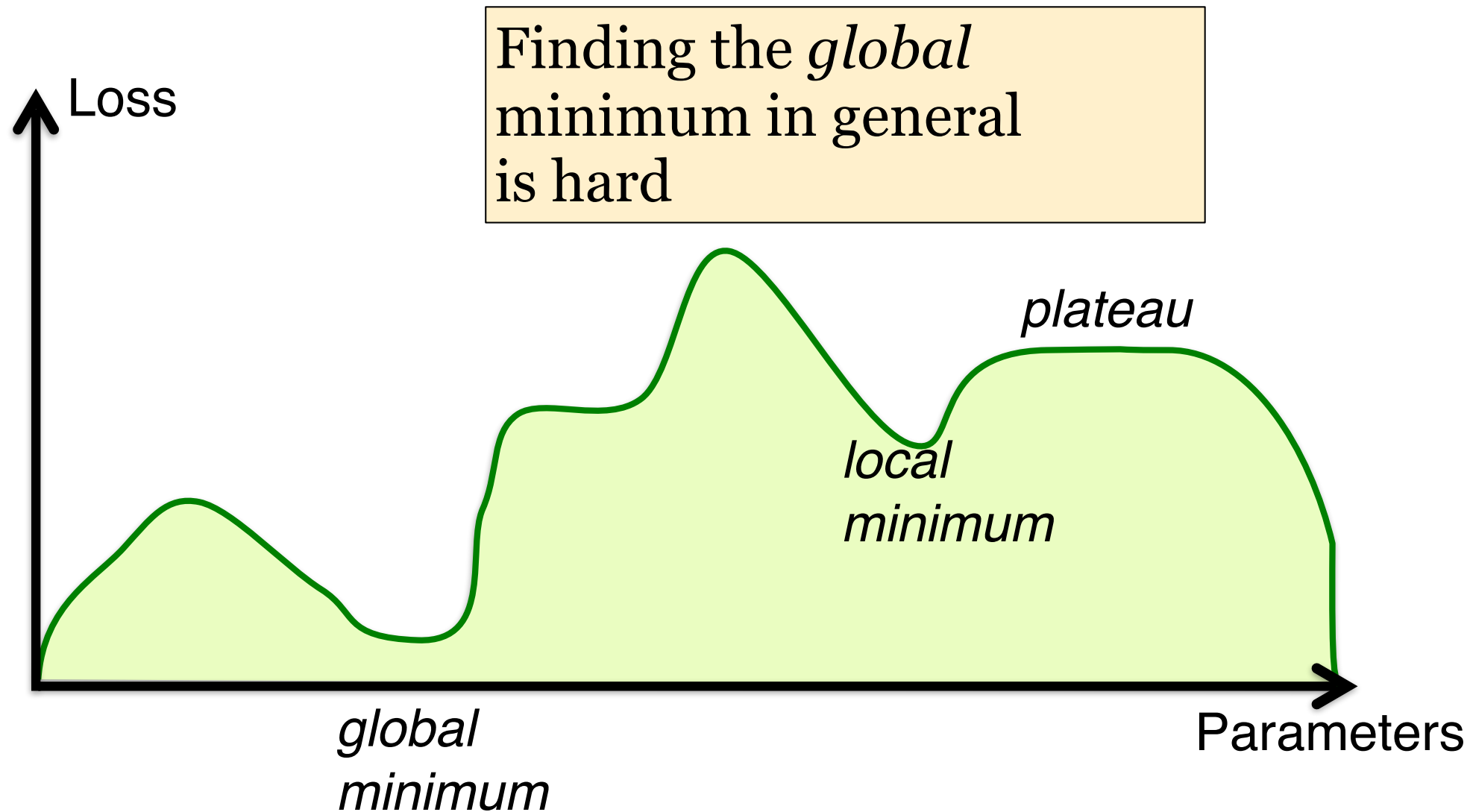
The *negative log probability of the correct label* is a loss function:

$-\log(P(y_i | \mathbf{x}_i))$  is *largest* ( $+\infty$ ) when we assign *all probability to the wrong label*,

$-\log(P(y_i | \mathbf{x}_i))$  is *smallest* (0) when we assign *all probability to the correct label*.

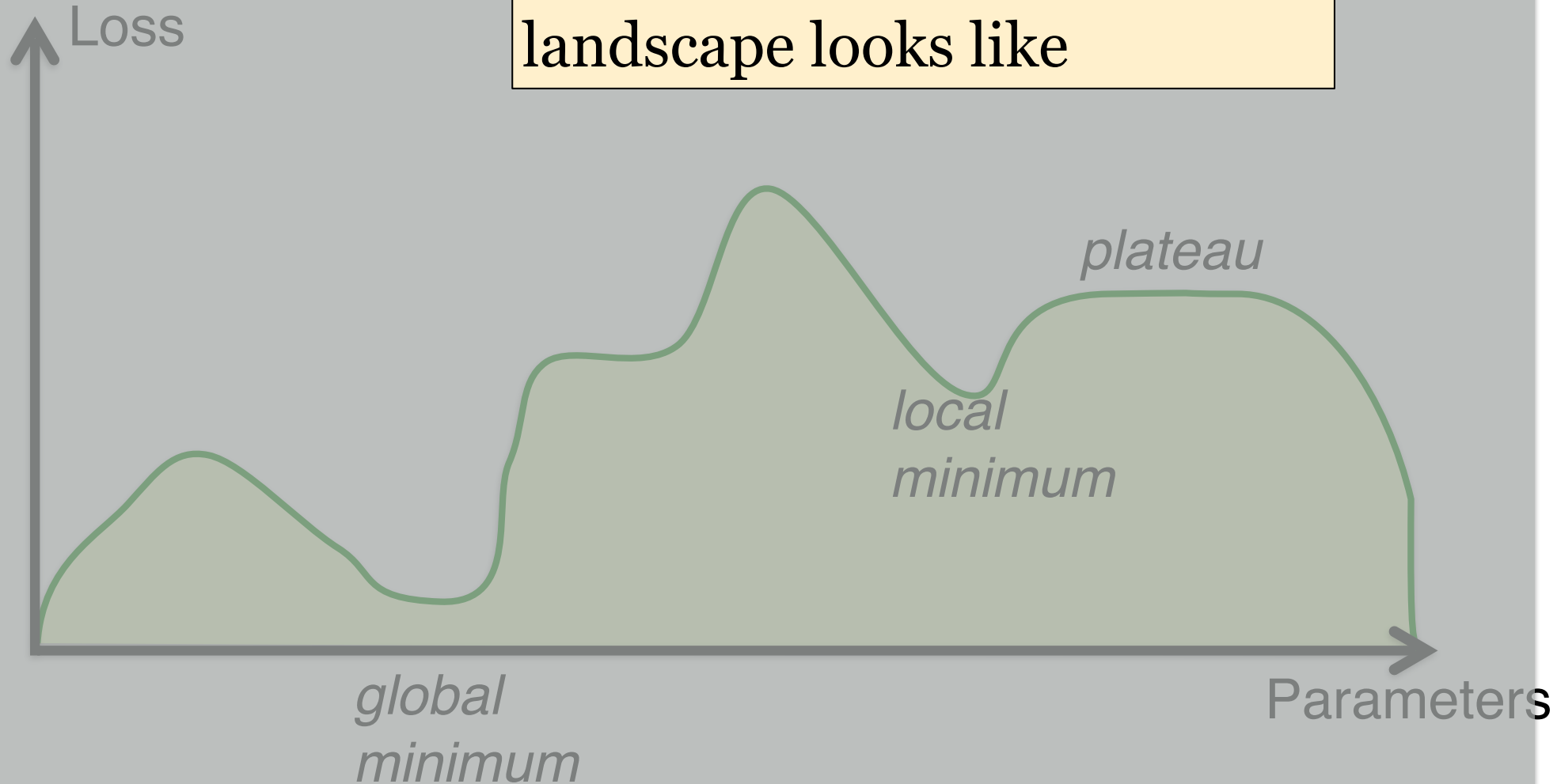
This *negative log likelihood loss* is also called *cross-entropy loss*

# The loss surface



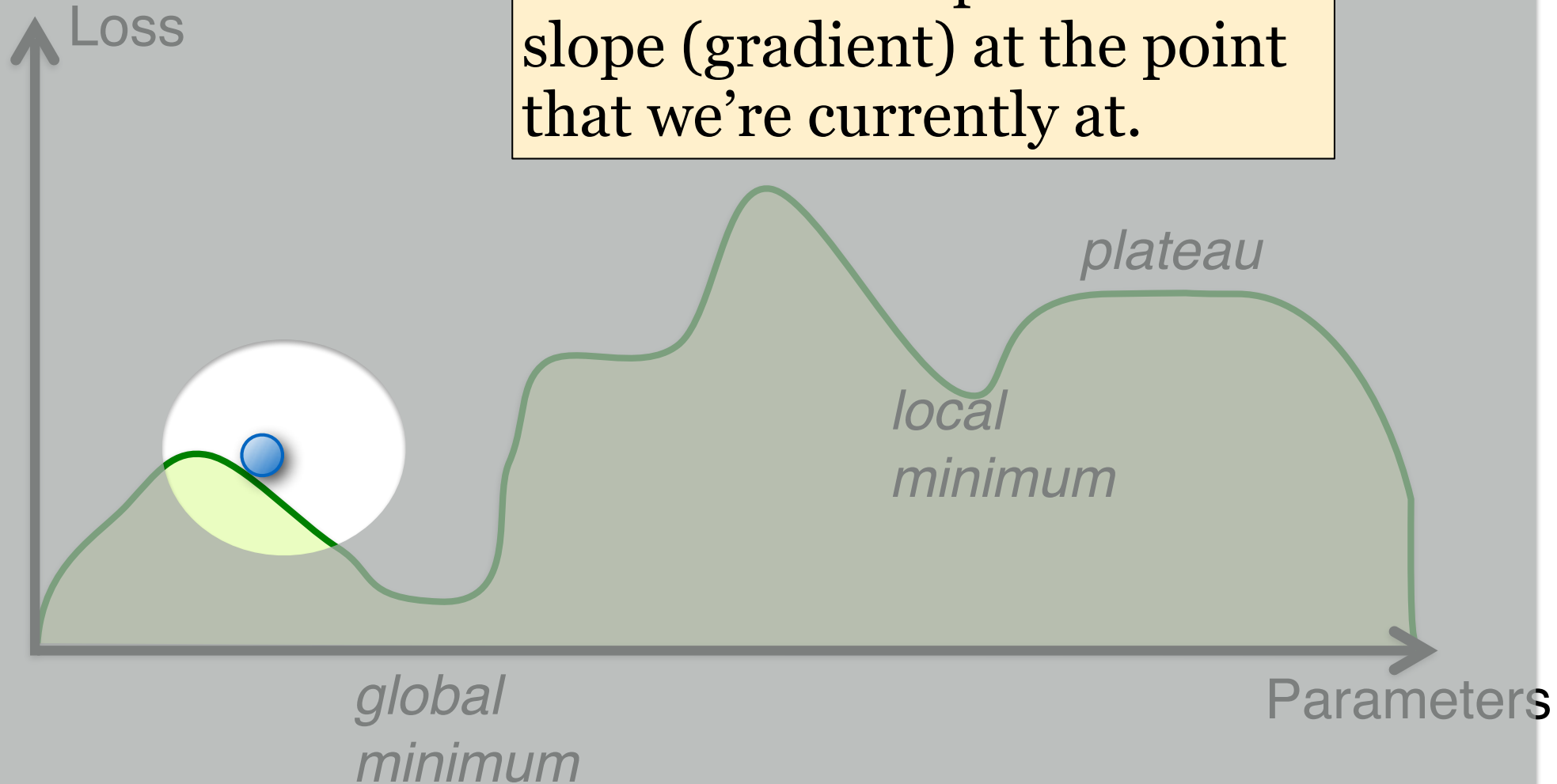
# Gradient of the loss

We don't even know how this landscape looks like

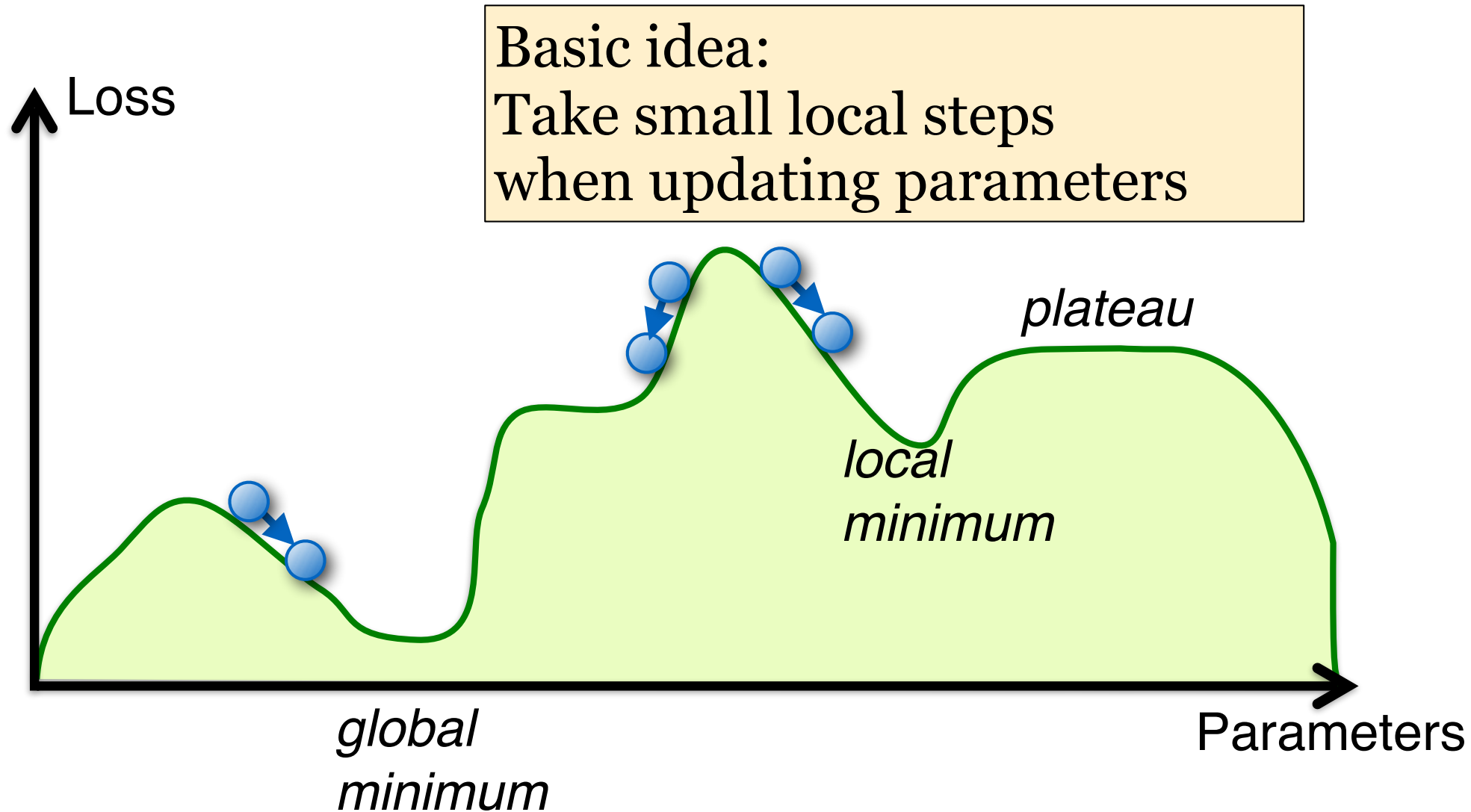


# Gradient of the loss

But we can compute the slope (gradient) at the point that we're currently at.



# Gradient descent



# (Stochastic) Gradient Descent

- We want to find **parameters that have minimal cost (loss)** on our training data.
- But we don't know the whole loss surface.
- However, the **gradient** of the cost (loss) of our current parameters tells us how the **slope of the loss surface** at the point given by our current parameters
- And then we can take a **(small) step in the right (downhill) direction** (to update our parameters)

## Gradient descent:

Compute loss for entire dataset before updating weights

## Stochastic gradient descent:

Compute loss for **one (randomly sampled) training example** before updating weights



# Neural Nets for NLP

# Neural Nets for NLP

Explain how to use a feedforward network for classification.

Explain how to use a feedforward network as a neural n-gram language model.

Discuss whether a one-hot encoding of the input is suitable for neural language models

Explain what a recurrent neural network is

# What are neural nets?

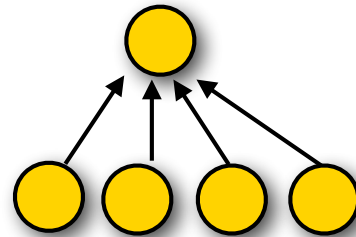
Simplest variant: single-layer feedforward net

For **binary**  
classification tasks:

**Single** output unit

Return 1 if  $y > 0.5$

Return 0 otherwise



**Output unit:** scalar  $y$

**Input layer:** vector  $\mathbf{x}$

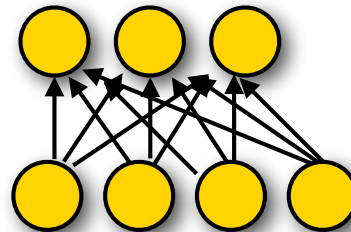
For **multiclass**  
classification tasks:

**K** output units (a vector)

Each output unit

$y_i = \text{class } i$

Return  $\text{argmax}_i(y_i)$



**Output layer:** vector  $\mathbf{y}$

**Input layer:** vector  $\mathbf{x}$

# Multiclass models: softmax( $y_i$ )

Multiclass classification = predict one of  $K$  classes.

Return the class  $i$  with the highest score:  $\operatorname{argmax}_i(y_i)$

In neural networks, this is typically done by using the **softmax** function, which maps real-valued vectors in  $\mathbb{R}^N$  into a distribution over the  $N$  outputs

For a vector  $\mathbf{z} = (z_0 \dots z_K)$ :  $P(i) = \operatorname{softmax}(z_i) = \exp(z_i) / \sum_{k=0..K} \exp(z_k)$   
This is just logistic regression

# Single-layer feedforward networks

## Single-layer (linear) feedforward network

$$y = \mathbf{w}\mathbf{x} + b \text{ (binary classification)}$$

$\mathbf{w}$  is a weight vector,  $b$  is a bias term (a scalar)

This is just a linear classifier (aka Perceptron)  
(the output  $y$  is a linear function of the input  $\mathbf{x}$ )

## Single-layer non-linear feedforward networks:

Pass  $\mathbf{w}\mathbf{x} + b$  through a non-linear activation function,  
e.g.  $y = \tanh(\mathbf{w}\mathbf{x} + b)$

# Nonlinear activation functions

**Sigmoid (logistic function):**  $\sigma(x) = 1/(1 + e^{-x})$

Useful for output units (probabilities) [0,1] range

**Hyperbolic tangent:**  $\tanh(x) = (e^{2x} - 1)/(e^{2x} + 1)$

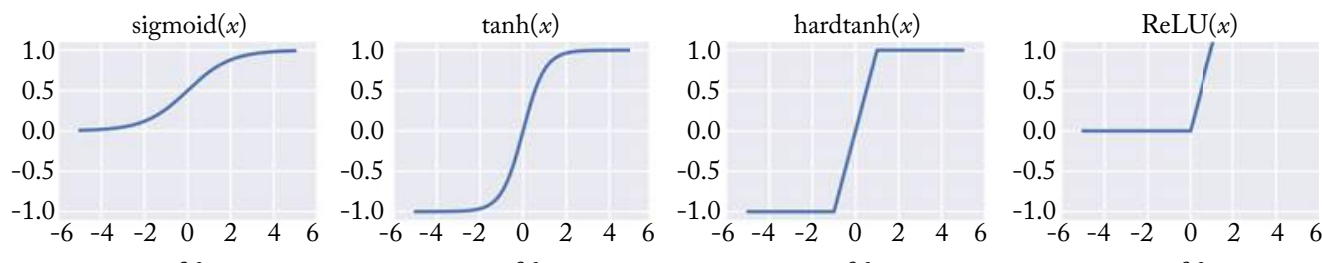
Useful for internal units: [-1,1] range

**Hard tanh (approximates tanh)**

$\text{htanh}(x) = -1$  for  $x < -1$ ,  $1$  for  $x > 1$ ,  $x$  otherwise

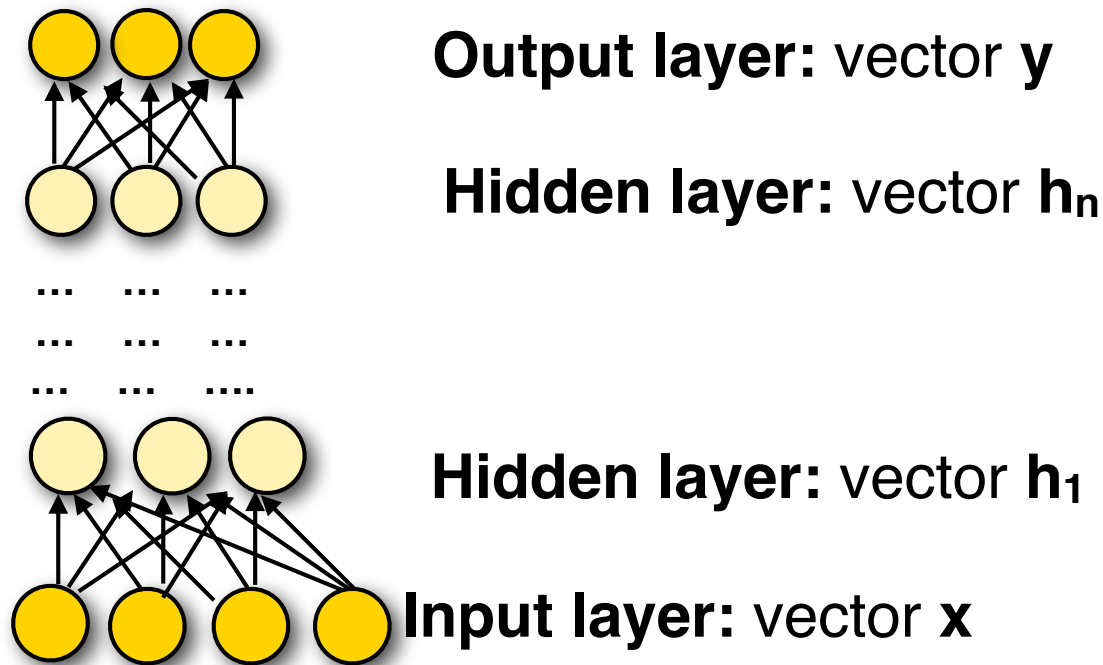
**Rectified Linear Unit:**  $\text{ReLU}(x) = \max(0, x)$

Useful for internal units



# Multi-layer feedforward networks

We can generalize this to multi-layer feedforward nets



# An $n$ -gram model $P(w \mid w_1 \dots w_k)$ as a feedforward net (naively)

- The **vocabulary**  $V$  contains  $n$  types (incl. UNK, BOS, EOS)
- We want to condition each word on  $k$  preceding words
- **[Naive]** Each **input word**  $w_i \in V$  (that we're conditioning on) is an  **$n$ -dimensional one-hot vector**  $v(w) = (0, \dots, 0, 1, 0, \dots, 0)$
- Our **input layer**  $\mathbf{x} = [v(w_1), \dots, v(w_k)]$  has  $n \times k$  elements
- To predict the probability over output words, the **output layer** is a softmax over  $n$  elements

$$P(w \mid w_1 \dots w_k) = \text{softmax}(\mathbf{h}\mathbf{W}^2 + \mathbf{b}^2)$$

With (say) one hidden layer  $\mathbf{h}$  we'll need two sets of parameters, one for  $\mathbf{h}$  and one for the output



# Naive neural n-gram model

Advantage over non-neural n-gram model:

- The hidden layer captures interactions among context words
- Increasing the order of the n-gram requires only a small linear increase in the number of parameters.

$\dim(\mathbf{W}^1)$  goes from  $k \cdot \dim(\text{emb}) \times \dim(\mathbf{h})$  to  $(k+1) \cdot \dim(\text{emb}) \times \dim(\mathbf{h})$

- Increasing the vocabulary also leads only to a linear increase in the number of parameters

But: with a one-hot encoding and  $\dim(V) \approx 10\text{K}$  or so, this model still requires a LOT of parameters to learn.

#parameters going to hidden layer:  $k \cdot \dim(V) \cdot \dim(\mathbf{h})$ ,

with  $\dim(\mathbf{h}) = 300$ ,  $\dim(V) = 10,000$  and  $k=3$ : 9,000,000

Plus #parameters going to output layer:  $\dim(\mathbf{h}) \cdot \dim(V)$

with  $\dim(\mathbf{h}) = 300$ ,  $\dim(V) = 10,000$ : 3,000,000

# Neural n-gram models

Naive neural language models have similar shortcomings to standard n-gram models

- Models get very large (and sparse) as n increases
- We can't generalize across similar contexts
- Markov (independence) assumptions in n-gram models are too strict

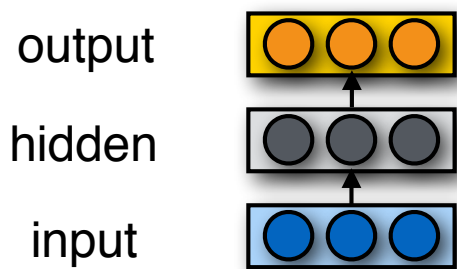
Solutions offered by less naive neural models:

- Do not represent context words as distinct, discrete symbols (i.e. very high-dimensional one-hot vectors), but use a dense low-dimensional vector representation where similar words have similar vectors [next class]
- Use recurrent nets that can encode variable-lengths contexts [later class]

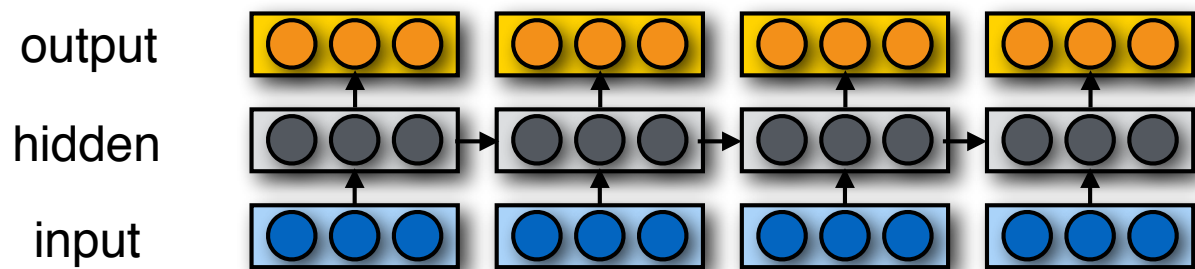
# Recurrent neural networks (RNNs)

**Basic RNN:** Modify the standard feedforward architecture (which predicts a string  $w_0 \dots w_n$  one word at a time) such that the output of the current step ( $w_i$ ) is given as additional input to the next time step (when predicting the output for  $w_{i+1}$ ).

“Output” — typically (the last) hidden layer.



**Feedforward Net**



**Recurrent Net**

# RNNs for language modeling

If our vocabulary consists of  $V$  words, the output layer (at each time step) has  $V$  units, one for each word.

The softmax gives a distribution over the  $V$  words for the next word.

To compute the probability of a string  $w_0 w_1 \dots w_n w_{n+1}$  (where  $w_0 = \langle s \rangle$ , and  $w_{n+1} = \langle \backslash s \rangle$ ), feed in  $w_i$  as input at time step  $i$  and compute

$$\prod_{i=1..n+1} P(w_i | w_0 \dots w_{i-1})$$

# Vector Semantics and Word Embeddings

# Vector Semantics and Word Embeddings

Describe the distributional hypothesis.

Explain how to represent words as vectors that capture distributional similarities

Describe how the vectors obtained from word embeddings like word2vec differ from vectors obtained via distributional approaches.

What training data is used for a skipgram classifier?

# Different approaches to lexical semantics

## Lexicographic tradition:

- Use lexicons, thesauri, ontologies
- Assume words have discrete word senses:  
bank1 = financial institution; bank2 = river bank, etc.
- May capture explicit relations between word (senses):  
“dog” is a “mammal”, etc.

## Distributional tradition:

- Map words to (sparse) vectors that capture corpus statistics
- Contemporary variant: use neural nets to learn dense vector  
“embeddings” from very large corpora  
(this is a prerequisite for most neural approaches to NLP)
- If each word type is mapped to a single vector, this ignores  
the fact that words have multiple senses or parts-of-speech

# The Distributional Hypothesis

Zellig Harris (1954):

*“oculist and eye-doctor ... occur in almost the same environments”*

*“If A and B have almost identical environments we say that they are synonyms.”*

John R. Firth 1957:

*You shall know a word by the company it keeps.*

The **contexts** in which a word appears tells us a lot about what it means.

Words that appear in similar contexts have similar meanings



# Two ways NLP uses context for semantics

## Distributional similarities (vector-space semantics):

Use the set of contexts in which words (= word types) appear to measure their similarity

Assumption: Words that appear in similar contexts (*tea, coffee*) have similar meanings.

## Word sense disambiguation (future lecture)

Use the context of a particular occurrence of a word (token) to identify which sense it has.

Assumption: If a word has multiple distinct senses (e.g. *plant: factory or green plant*), each sense will appear in different contexts.

# Distributional Similarities

Measure the **semantic similarity** of words in terms of the **similarity of the contexts** in which the words appear

Represent words as vectors such that

- each **vector element** (dimension) corresponds to a different **context**
- the vector for any particular word captures how **strongly it is associated** with each context

Compute the semantic similarity of words as the **similarity of their vectors**.

# What is a ‘context’?

There are many different definitions of context that yield different kinds of similarities:

## Contexts defined by nearby words:

How often does  $w$  appear near the word *drink*?

Near = “*drink* appears within a window of  $\pm k$  words of  $w$ ”,  
or “*drink* appears in the same document/sentence as  $w$ ”

This yields fairly broad thematic similarities.

## Contexts defined by grammatical relations:

How often is (the noun)  $w$  used as the subject (object) of the verb *drink*? (Requires a parser).

This gives more fine-grained similarities.

# Vector representations of words

“Traditional” **distributional similarity** approaches represent words as **sparse vectors**

- Each dimension represents one specific context
- Vector entries are based on word-context co-occurrence statistics (counts or PMI values)

Alternative, **dense vector** representations:

- We can use Singular Value Decomposition to turn these sparse vectors into dense vectors (Latent Semantic Analysis)
- We can also use **neural** models to explicitly learn a dense vector representation (**embedding**) (word2vec, Glove, etc.)

**Sparse** vectors = **most entries are zero**

**Dense** vectors = **most entries are non-zero**

# Word2Vec Embeddings

## Main idea:

Use a **binary classifier** to predict which words appear in the context of (i.e. near) a target word.

The **parameters of that classifier** provide a dense vector representation of the target word (embedding)

Words that appear in similar contexts (that have high distributional similarity) will have very similar vector representations.

These models can be trained on large amounts of raw text (and pre-trained embeddings can be downloaded)

# Skip-Gram with negative sampling

Train a binary classifier that decides whether a target word  $t$  appears in the context of other words  $c_{1..k}$

- **Context**: the set of  $k$  words near (surrounding)  $t$
- Treat the target word  $t$  and any word that *actually* appears in its context in a real corpus as **positive** examples
- Treat the target word  $t$  and *randomly sampled* words that don't appear in its context as **negative** examples
- Train a **binary logistic regression** classifier to distinguish these cases
- The **weights** of this classifier depend on the **similarity** of  $t$  and the words in  $c_{1..k}$

Use the weights of this classifier as embeddings for  $t$

# The Skip-Gram classifier

Use **logistic regression** to predict whether the pair  $(t, c)$  (target word  $t$  and a context word  $c$ ) is a positive or negative example:

$$P(+|t, c) = \frac{1}{1 + e^{-t \cdot c}} \quad P(-|t, c) = 1 - P(+|t, c) = \frac{e^{-t \cdot c}}{1 + e^{-t \cdot c}}$$

Assume that **t and c are represented as vectors**, so that their dot product  $tc$  captures their similarity

# Summary: How to learn word2vec (skip-gram) embeddings

For a vocabulary of size  $V$ : Start with  $V$  random 300-dimensional vectors as initial embeddings

Train a logistic regression classifier to distinguish words that co-occur in corpus from those that don't

- Pairs of words that co-occur are positive examples

- Pairs of words that don't co-occur are negative examples

- Train the classifier to distinguish these by slowly adjusting all the embeddings to improve the classifier performance

Throw away the classifier code and keep the embeddings.



# POS tagging and sequence labeling

# POS tagging and sequence labeling

Why has POS tagging been seen as an important step in the NLP pipeline?

Discuss the advantages and disadvantages of a very coarse POS tag set vs. a very fine grained one.

Define a bigram HMM model.

Explain the Viterbi algorithm for POS tagging with a bigram HMM.

Explain how to frame named entity recognition as a sequence labeling task

Explain the advantages of discriminative models for sequence labeling

# POS Tagging

Words often have more than one POS:

- *The **back** door* (adjective)
- *On my **back*** (noun)
- *Win the voters **back*** (particle)
- *Promised to **back** the bill* (verb)

The POS tagging task is to determine the POS tag for a particular instance of a word.

Since there is ambiguity, we cannot simply look up the correct POS in a dictionary.

These examples from Dekang Lin

# Why POS tagging?

POS tagging is traditionally viewed as a prerequisite for further analysis:

## –Speech synthesis:

How to pronounce “lead”?

INsult or inSULT, OBject or obJECT, OVERflow or overFLOW,  
DIScount or disCOUNT, CONtent or conTENT

## –Parsing:

What words are in the sentence?

## –Information extraction:

Finding names, relations, etc.

## –Machine Translation:

The noun “content” may have a different translation from the adjective.

# Defining an annotation scheme

Training and evaluating models for these NLP tasks requires large **corpora annotated with the desired representations.**

Annotation at scale is expensive, so **a few existing corpora** and their **annotations** and **annotation schemes** (tag sets, etc.) often become the de facto standard for the field.

It is difficult to know what the ‘right’ annotation scheme should be for any particular task

How difficult is it to achieve high accuracy for that annotation?

How useful is this annotation scheme for downstream tasks in the pipeline?

⇒ We often can't know the answer until we've annotated a lot of data...

# Evaluation metric: test accuracy

**How many words in the unseen test data can you tag correctly?**

State of the art on Penn Treebank: around 97%.

⇒ **How many *sentences* can you tag correctly?**

Compare your model against a **baseline**

Standard: assign to each word its most likely tag  
(use training corpus to estimate  $P(\text{tlw})$  )

Baseline performance on Penn Treebank: around 93.7%

... and a **(human) ceiling**

How often do human annotators agree on the same tag?

Penn Treebank: around 97%

# Qualitative evaluation

Generate a **confusion matrix** (for development data):  
How often was a word with tag *i* mistagged as tag *j*:

		Correct Tags						
		IN	JJ	NN	NNP	RB	VBD	VBN
Predicted Tags	IN	—	.2			.7		
	JJ	.2	—	3.3	2.1	1.7	.2	2.7
	NN		8.7	—				.2
	NNP	.2	3.3	4.1	—	.2		
	RB	2.2	2.0	.5		—		
	VBD		.3	.5			—	4.4
	VBN		2.8				2.6	—

% of errors caused by mistagging VBN as JJ

See what errors are causing problems:

- Noun (NN) vs ProperNoun (NNP) vs Adj (JJ)
- Preterite (VBD) vs Participle (VBN) vs Adjective (JJ)

# POS tagging with generative models

$$\begin{aligned}\operatorname{argmax}_{\mathbf{t}} P(\mathbf{t}|\mathbf{w}) &= \operatorname{argmax}_{\mathbf{t}} \frac{P(\mathbf{t}, \mathbf{w})}{P(\mathbf{w})} \\ &= \operatorname{argmax}_{\mathbf{t}} P(\mathbf{t}, \mathbf{w}) \\ &= \operatorname{argmax}_{\mathbf{t}} P(\mathbf{t})P(\mathbf{w}|\mathbf{t})\end{aligned}$$

$P(\mathbf{t}, \mathbf{w})$ : the joint distribution of the labels we want to predict ( $\mathbf{t}$ ) and the observed data ( $\mathbf{w}$ ).

We decompose  $P(\mathbf{t}, \mathbf{w})$  into  $P(\mathbf{t})$  and  $P(\mathbf{w} | \mathbf{t})$  since these distributions are easier to estimate.

Models based on joint distributions of labels and observed data are called **generative models**: think of  $P(\mathbf{t})P(\mathbf{w} | \mathbf{t})$  as a stochastic process that first generates the labels, and then generates the data we see, based on these labels.



# Hidden Markov Models (HMMs)

HMMs are the most commonly used generative models for POS tagging (and other tasks, e.g. in speech recognition)

HMMs make specific **independence assumptions** in  $P(\mathbf{t})$  and  $P(\mathbf{w} | \mathbf{t})$ :

1)  $P(\mathbf{t})$  is an n-gram (typically bigram or trigram) model over tags:

$$P_{\text{bigram}}(\mathbf{t}) = \prod_i P(t^{(i)} | t^{(i-1)}) \quad P_{\text{trigram}}(\mathbf{t}) = \prod_i P(t^{(i)} | t^{(i-1)}, t^{(i-2)})$$

$P(t^{(i)} | t^{(i-1)})$  and  $P(t^{(i)} | t^{(i-1)}, t^{(i-2)})$  are called **transition probabilities**

2) In  $P(\mathbf{w} | \mathbf{t})$ , each  $w^{(i)}$  depends only on [is generated by/conditioned on]  $t^{(i)}$ :

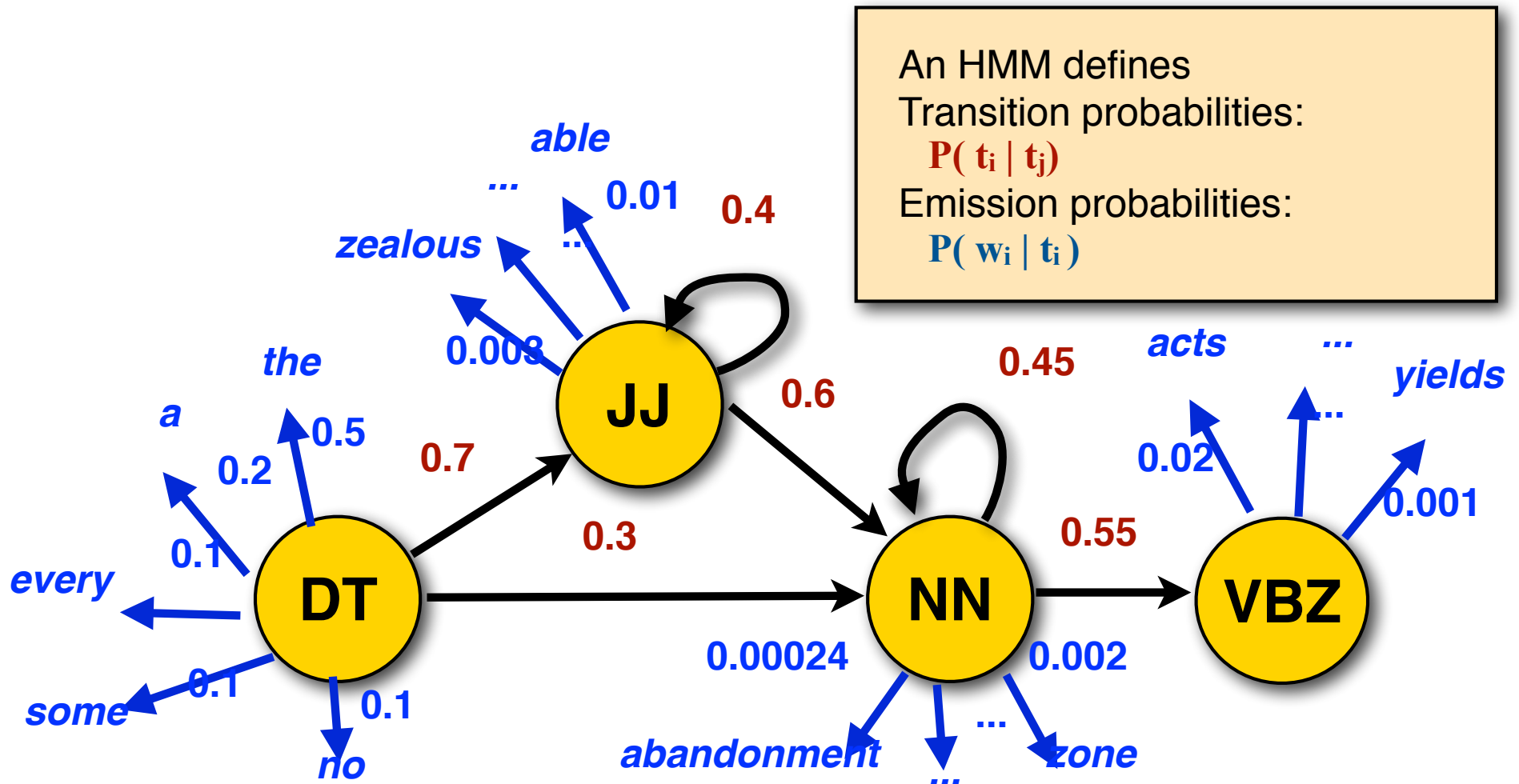
$$P(\mathbf{w} | \mathbf{t}) = \prod_i P(w^{(i)} | t^{(i)})$$

$P(w^{(i)} | t^{(i)})$  are called **emission probabilities**

These probabilities don't depend on the string position  $(i)$ , but are defined over word and tag types.

With subscripts  $i, j, k$ , to index types, they become  $P(t_i | t_j)$ ,  $P(t_i | t_j, t_k)$ ,  $P(w_i | t_j)$

# HMMs as probabilistic automata



# Learning an HMM from *labeled* data

```
Pierre_NNP Vinken_NNP ,_, 61_CD years_NNS  
old_JJ ,_, will_MD join_VB the_DT board_NN  
as_IN a_DT nonexecutive_JJ director_NN Nov._NNP  
29_CD ._. 
```

We **count** how often we see  $t_i t_j$  and  $w_j | t_i$  etc. in the data (use relative frequency estimates):

Learning the transition probabilities:

$$P(t_j | t_i) = \frac{C(t_i t_j)}{C(t_i)}$$

Learning the emission probabilities:

$$P(w_j | t_i) = \frac{C(w_j | t_i)}{C(t_i)}$$

# HMM decoding (Viterbi)

We observe a sentence  $\mathbf{w} = w^{(1)} \dots w^{(N)}$

$\mathbf{w} =$  “*she promised to back the bill*”

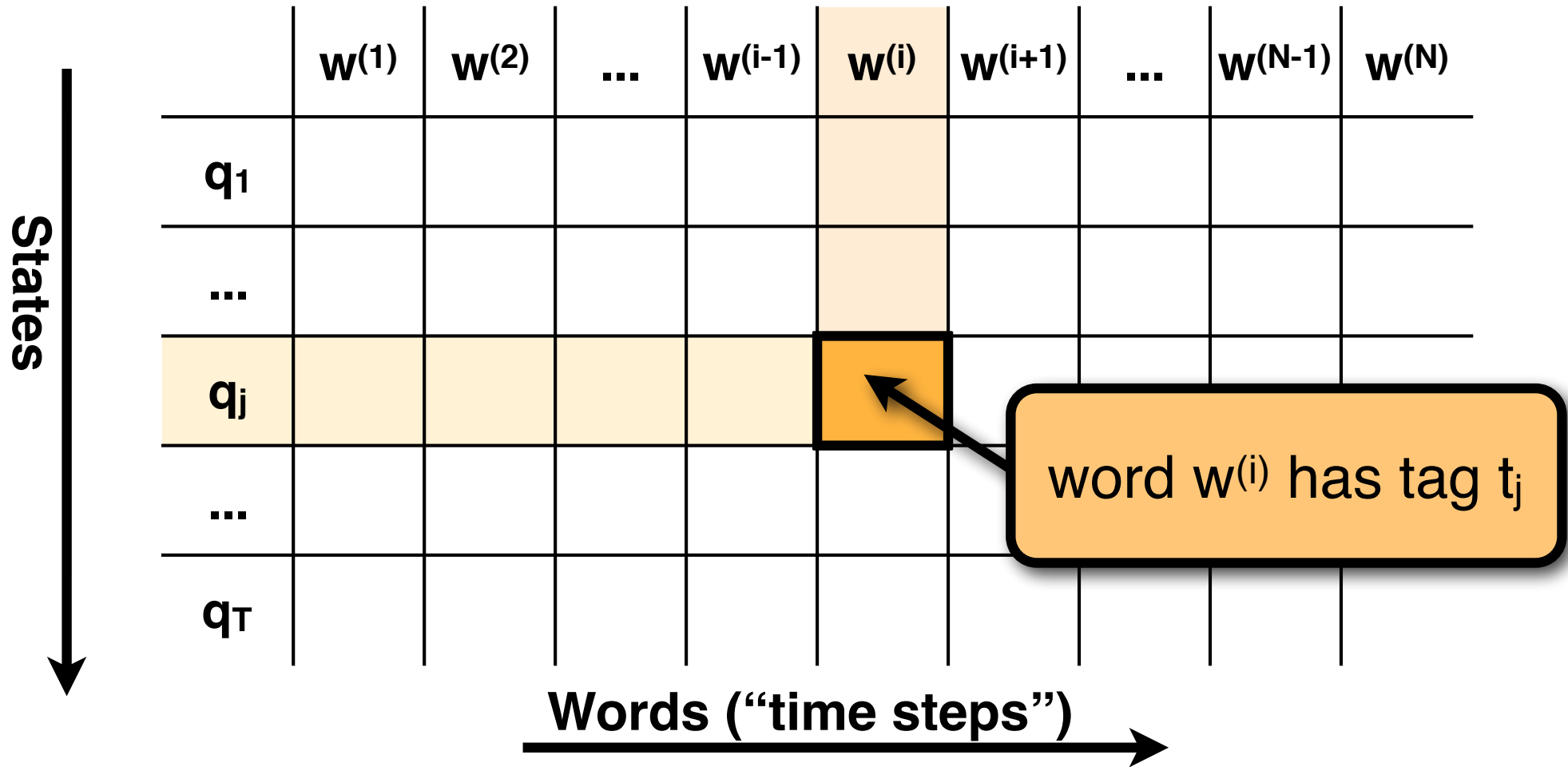
We want to use an HMM tagger to find its POS tags  $\mathbf{t}$

$$\mathbf{t}^* = \operatorname{argmax}_{\mathbf{t}} P(\mathbf{w}, \mathbf{t})$$

$$= \operatorname{argmax}_{\mathbf{t}} P(t^{(1)}) \cdot P(w^{(1)} | t^{(1)}) \cdot P(t^{(2)} | t^{(1)}) \cdot \dots \cdot P(w^{(N)} | t^{(N)})$$

To do this efficiently, we will use a **dynamic programming** technique called the Viterbi algorithm which exploits the **independence assumptions** in the HMM.

# Bookkeeping: the trellis



We use a  $N \times T$  table (“**trellis**”) to keep track of the HMM. The HMM can assign one of the  $T$  tags to each of the  $N$  words.

# Using the trellis to find $t^*$

Let  $\text{trellis}[i][j]$  (word  $w^{(i)}$  and tag  $t_j$ ) store the probability of the **best** tag sequence for  $w^{(1)} \dots w^{(i)}$  that ends in  $t_j$

$$\text{trellis}[i][j] =_{\text{def}} \max P(w^{(1)} \dots w^{(i)}, t^{(1)} \dots, t^{(i)} = t_j)$$

For each cell  $\text{trellis}[i][j]$ , we find the **best cell in the previous column** ( $\text{trellis}[i-1][k^*]$ ) based on the entries in the **previous column** and the **transition probabilities**  $P(t_j | t_k)$

$$k^* \text{ for } \text{trellis}[i][j] := \text{Max}_k ( \text{trellis}[i-1][k] \cdot P(t_j | t_k) )$$

The entry in  $\text{trellis}[i][j]$  includes the emission probability  $P(w^{(i)} | t_j)$

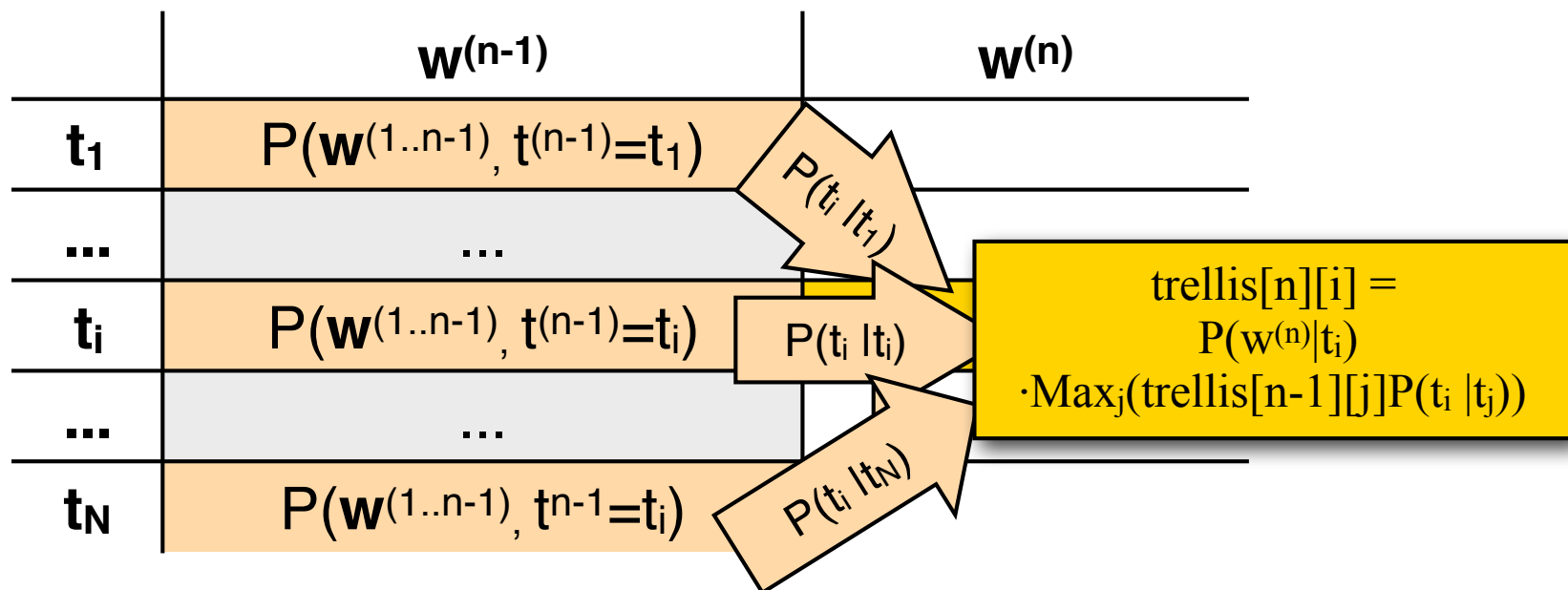
$$\text{trellis}[i][j] := P(w^{(i)} | t_j) \cdot ( \text{trellis}[i-1][k^*] \cdot P(t_j | t_{k^*}) )$$

We also associate a **backpointer** from  $\text{trellis}[i][j]$  to  $\text{trellis}[i-1][k^*]$

Finally, we pick the **highest scoring entry in the last column** of the trellis (= for the last word) and follow the backpointers

# At any internal cell

- For each cell in the preceding column: multiply its entry with the transition probability to the current cell.
- Keep a single backpointer to the best (highest scoring) cell in the preceding column
- Multiply this score with the emission probability of the current word

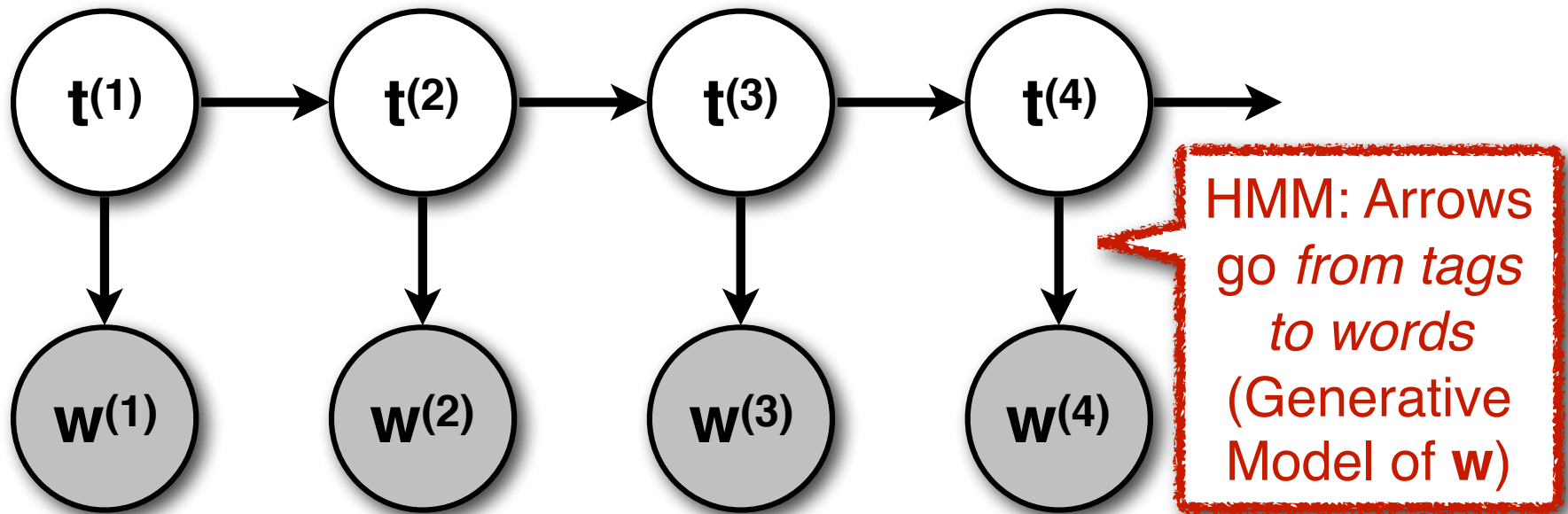


# HMMs as graphical models

HMMs are **generative models** of the observed string  $\mathbf{w}$

They 'generate'  $\mathbf{w}$  with  $P(\mathbf{w}, \mathbf{t}) = \prod_i P(t^{(i)} | t^{(i-1)}) P(w^{(i)} | t^{(i)})$

When we use an HMM for tagging,  
we observe  $\mathbf{w}$ , and need to find  $\mathbf{t}$

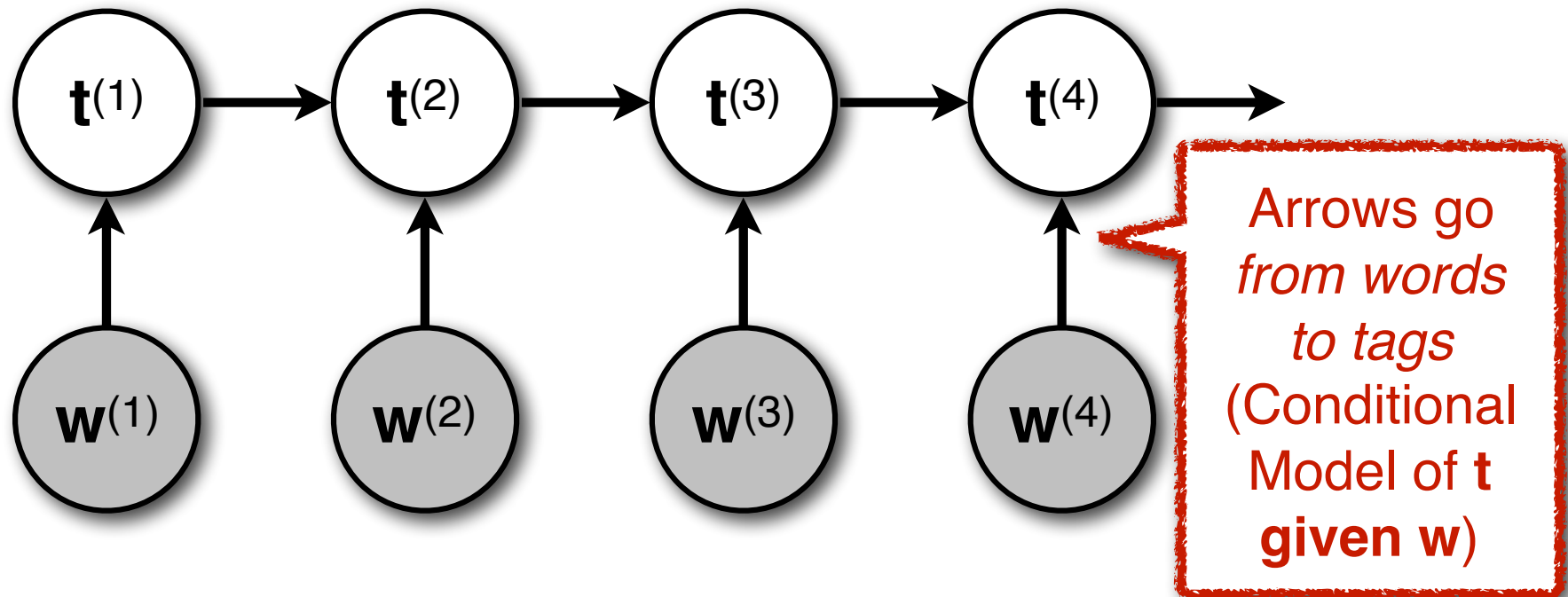




# Discriminative probability models

A discriminative or **conditional** model of the labels  $\mathbf{t}$  given the observed input string  $\mathbf{w}$  models

$P(\mathbf{t} | \mathbf{w}) = \prod_i P(t^{(i)} | w^{(i)}, t^{(i-1)})$  directly.



# Discriminative models

There are two main types of discriminative probability models:

- Maximum Entropy Markov Models (MEMMs)
- Conditional Random Fields (CRFs)

MEMMs and CRFs:

- are both based on logistic regression
- have the same graphical model
- require the Viterbi algorithm for tagging
- differ in that MEMMs consist of independently learned distributions, while CRFs are trained to maximize the probability of the entire sequence

# Advantages of discriminative models

**We're usually not really interested in  $P(\mathbf{w} | \mathbf{t})$ .**

–  $\mathbf{w}$  is given. We don't need to predict it!

Why not model what we're actually interested in:  $P(\mathbf{t} | \mathbf{w})$

**Modeling  $P(\mathbf{w} | \mathbf{t})$  well is quite difficult:**

– Prefixes (capital letters) or suffixes are good predictors for certain classes of  $\mathbf{t}$  (proper nouns, adverbs,...)

– So we don't want to model words as atomic symbols, but in terms of features

– These features may also help us deal with unknown words

– But features may not be independent

**Modeling  $P(\mathbf{t} | \mathbf{w})$  with features should be easier:**

– Now we can incorporate arbitrary features of the word, because we don't need to predict  $\mathbf{w}$  anymore