

# Statistical NLP

## Spring 2008



### Lecture 5: WSD / Maxent

Dan Klein – UC Berkeley

## Word Senses

- Words have multiple distinct meanings, or senses:
  - Plant: living plant, manufacturing plant, ...
  - Title: name of a work, ownership document, form of address, material at the start of a film, ...
- Many levels of sense distinctions
  - Homonymy: totally unrelated meanings (river bank, money bank)
  - Polysemy: related meanings (star in sky, star on tv)
  - Systematic polysemy: productive meaning extensions (organizations to their buildings) or metaphor
  - Sense distinctions can be extremely subtle (or not)
- Granularity of senses needed depends a lot on the task
- Why is it important to model word senses?
  - Translation, parsing, information retrieval?

## Word Sense Disambiguation

- Example: living plant vs. manufacturing plant
- How do we tell these senses apart?
  - "context"

The manufacturing **plant** which had previously sustained the town's economy shut down after an extended labor strike.

  - Maybe it's just text categorization
  - Each word sense represents a topic
  - Run the naive-bayes classifier from last class?
- Bag-of-words classification works ok for noun senses
  - 90% on classic, shockingly easy examples (line, interest, star)
  - 80% on senseval-1 nouns
  - 70% on senseval-1 verbs

## Verb WSD

- Why are verbs harder?
  - Verbal senses less topical
  - More sensitive to structure, argument choice
- Verb Example: "Serve"
  - [function] The tree stump serves as a table
  - [enable] The scandal served to increase his popularity
  - [dish] We serve meals for the homeless
  - [enlist] He served his country
  - [jail] He served six years for embezzlement
  - [tennis] It was Agassi's turn to serve
  - [legal] He was served by the sheriff

## Various Approaches to WSD

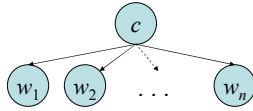
- Unsupervised learning
  - Bootstrapping (Yarowsky 95)
  - Clustering
- Indirect supervision
  - From thesauri
  - From WordNet
  - From parallel corpora
- Supervised learning
  - Most systems do some kind of supervised learning
  - Many competing classification technologies perform about the same (it's all about the knowledge sources you tap)
  - Problem: training data available for only a few words

## Resources

- WordNet
  - Hand-build (but large) hierarchy of word senses
  - Basically a hierarchical thesaurus
- SensEval
  - A WSD competition, of which there have been 3 iterations
  - Training / test sets for a wide range of words, difficulties, and parts-of-speech
  - Bake-off where lots of labs tried lots of competing approaches
- SemCor
  - A big chunk of the Brown corpus annotated with WordNet senses
- OtherResources
  - The Open Mind Word Expert
  - Parallel texts
  - Flat thesauri

## Knowledge Sources

- So what do we need to model to handle “serve”?
  - There are distant topical cues
    - ... point ... court ..... serve ..... game ...



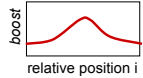
$$P(c, w_1, w_2, \dots, w_n) = P(c) \prod_i P(w_i | c)$$

## Weighted Windows with NB

- Distance conditioning
  - Some words are important only when they are nearby
  - ... as ... point ... court ..... serve ..... game ...
  - ... court ..... serve as ..... game ..... point

$$P(c, w_{-k}, \dots, w_{-1}, w_0, w_{+1}, \dots, w_{+k}) = P(c) \prod_{i=-k}^{+k} P(w_i | c, bin(i))$$

- Distance weighting
  - Nearby words should get a larger vote
  - ... court ..... serve as ..... game ..... point



$$P(c, w_{-k}, \dots, w_{-1}, w_0, w_{+1}, \dots, w_{+k}) = P(c) \prod_{i=-k}^{+k} P(w_i | c)^{boost(i)}$$

## Better Features

- There are smarter features:
  - Argument selectional preference:
    - serve NP[meals] vs. serve NP[papers] vs. serve NP[country]
  - Subcategorization:
    - [function] serve PP[as]
    - [enable] serve VP[to]
    - [tennis] serve <intransitive>
    - [food] serve NP {PP[to]}
  - Can capture poorly (but robustly) with local windows
  - ... but we can also use a parser and get these features explicitly
- Other constraints (Yarowsky 95)
  - One-sense-per-discourse (only true for broad topical distinctions)
  - One-sense-per-collocation (pretty reliable when it kicks in: manufacturing plant, flowering plant)

## Complex Features with NB?

- Example: Washington County jail served 11,166 meals last month - a figure that translates to feeding some 120 people three times daily for 31 days.
- So we have a decision to make based on a set of cues:
  - context:jail, context:county, context:feeding, ...
  - local-context:jail, local-context:meals
  - subcat:NP, direct-object-head:meals
- Not clear how build a generative derivation for these:
  - Choose topic, then decide on having a transitive usage, then pick “meals” to be the object’s head, then generate other words?
  - How about the words that appear in multiple features?
  - Hard to make this work (though maybe possible)
  - No real reason to try

## A Discriminative Approach

- View WSD as a discrimination task (regression, really)
  - $P(\text{sense} | \text{context:jail, context:county, context:feeding, ... local-context:jail, local-context:meals subcat:NP, direct-object-head:meals, ...})$
- Have to estimate multinomial (over senses) where there are a huge number of things to condition on
  - History is too complex to think about this as a smoothing / back-off problem
- Many feature-based classification techniques out there
- We tend to need ones that output distributions over classes (why?)

## Feature Representations

- Input  $d$ : Washington County jail served 11,166 meals last month - a figure that translates to feeding some 120 people three times daily for 31 days.
- Output  $\{f_i(d)\}$ :
  - context:jail = 1
  - context:county = 1
  - context:feeding = 1
  - context:game = 0
  - ...
  - local-context:jail = 1
  - local-context:meals = 1
  - ...
  - subcat:NP = 1
  - subcat:PP = 0
  - ...
  - object-head:meals = 1
  - object-head:ball = 0
- Features are indicator functions  $f_i$  which count the occurrences of certain patterns in the input
- We map each input to a vector of feature predicate counts

## Linear Classifiers

- For a pair  $(c, d)$ , we take a weighted vote for each class:

$$\text{vote}(c | d) = \exp \sum_i \lambda_i(c) f_i(d)$$

Feature	Food	Jail	Tennis
context:jail	-0.5 * 1	+1.2 * 1	-0.8 * 1
subcat:NP	+1.0 * 1	+1.0 * 1	-0.3 * 1
object-head:meals	+2.0 * 1	-1.5 * 1	-1.5 * 1
object-head:years = 0	-1.8 * 0	+2.1 * 0	-1.1 * 0
TOTAL	+3.5	+0.7	-2.6

- There are many ways to set these weights
  - Perceptron: find a currently misclassified example, and nudge weights in the direction of a correct classification
  - Other discriminative methods usually work in the same way: try out various weights until you maximize some objective

## Maximum-Entropy Classifiers

- Exponential (log-linear, maxent, logistic, Gibbs) models:
  - Turn the votes into a probability distribution:

$$P(c | d, \lambda) = \frac{\exp \sum_i \lambda_i(c) f_i(d)}{\sum_{c'} \exp \sum_i \lambda_i(c') f_i(d)}$$

← Makes votes positive.  
← Normalizes votes.

- For any weight vector  $\{\lambda_i\}$ , we get a conditional probability model  $P(c | d, \lambda)$ .
- We want to choose parameters that *maximize the conditional (log) likelihood* of the data:

$$\log P(C | D, \lambda) = \sum_{(c,d) \in (C,D)} \log P(c | d, \lambda) = \sum_{(c,d) \in (C,D)} \log \frac{\exp \sum_i \lambda_i(c) f_i(d)}{\sum_{c'} \exp \sum_i \lambda_i(c') f_i(d)}$$

## Building a Maxent Model

- How to define features:
  - Features are patterns in the input which we think the weighted vote should depend on
  - Usually features added incrementally to target errors
  - If we're careful, adding some mediocre features into the mix won't hurt (but won't help either)
- How to learn model weights?
  - Maxent just one method
  - Use a numerical optimization package
  - Given a current weight vector, need to calculate (repeatedly):
    - Conditional likelihood of the data
    - Derivative of that likelihood wrt each feature weight

## The Likelihood Value

- The (log) conditional likelihood is a function of the iid data  $(C, D)$  and the parameters  $\lambda$ :

$$\log P(C | D, \lambda) = \log \prod_{(c,d) \in (C,D)} P(c | d, \lambda) = \sum_{(c,d) \in (C,D)} \log P(c | d, \lambda)$$

- If there aren't many values of  $c$ , it's easy to calculate:

$$\log P(C | D, \lambda) = \sum_{(c,d) \in (C,D)} \log \frac{\exp \sum_i \lambda_i(c) f_i(d)}{\sum_{c'} \exp \sum_i \lambda_i(c') f_i(d)}$$

- We can separate this into two components:

$$\log P(C | D, \lambda) = \sum_{(c,d) \in (C,D)} \log \exp \sum_i \lambda_i(c) f_i(d) - \sum_{(c,d) \in (C,D)} \log \sum_{c'} \exp \sum_i \lambda_i(c') f_i(d)$$

$$\log P(C | D, \lambda) = N(\lambda) - M(\lambda)$$

## The Derivative I: Numerator

$$\begin{aligned} \frac{\partial N(\lambda)}{\partial \lambda_i(c)} &= \frac{\partial \sum_k \log \exp \sum_i \lambda_i(c_k) f_i(d_k)}{\partial \lambda_i(c)} = \frac{\partial \sum_k \lambda_i(c_k) f_i(d_k)}{\partial \lambda_i(c)} \\ &= \sum_{k: c_k=c} \frac{\partial \sum_i \lambda_i(c) f_i(d_k)}{\partial \lambda_i(c)} = \sum_{k: c_k=c} f_i(d_k) \end{aligned}$$

Derivative of the numerator is the empirical count ( $f_i, c$ )

E.g.: we actually saw the word "dish" with the "food" sense 3 times (maybe twice in one example and once in another).

## The Derivative II: Denominator

$$\begin{aligned} \frac{\partial M(\lambda)}{\partial \lambda_i(c)} &= \frac{\partial \sum_k \log \sum_{c'} \exp \sum_i \lambda_i(c') f_i(d_k)}{\partial \lambda_i(c)} \\ &= \sum_k \frac{1}{\sum_{c'} \exp \sum_i \lambda_i(c') f_i(d_k)} \frac{\partial \sum_{c'} \exp \sum_i \lambda_i(c') f_i(d_k)}{\partial \lambda_i(c)} \\ &= \sum_k \frac{1}{\sum_{c'} \exp \sum_i \lambda_i(c') f_i(d_k)} \sum_{c'} \frac{\exp \sum_i \lambda_i(c') f_i(d_k)}{1} \frac{\partial \sum_i \lambda_i(c') f_i(d_k)}{\partial \lambda_i(c)} \\ &= \sum_k \sum_{c'} \frac{\exp \sum_i \lambda_i(c') f_i(d_k)}{\sum_{c''} \exp \sum_i \lambda_i(c'') f_i(d_k)} \frac{\partial \sum_i \lambda_i(c') f_i(d_k)}{\partial \lambda_i(c)} \\ &= \sum_k P(c | d_k, \lambda) f_i(d_k) = \text{predicted count}(f_i, \lambda) \end{aligned}$$

## The Derivative III

$$\frac{\partial \log P(C|D, \lambda)}{\partial \lambda_i(c)} = \text{actual count}(f_i, c) - \text{predicted count}(f_i, \lambda)$$

	C	P(c   λ)
d	meal, jail, ...	food .4
	jail, term, ...	prison .8
	actual = 1	empirical = 1.2

The  $\lambda$ (prison) weight for the "context-word:jail" feature:

- The optimum parameters are the ones for which each feature's predicted expectation equals its empirical expectation. The optimum distribution is:
  - Always unique (but parameters may not be unique)
  - Always exists (if features counts are from actual data).

## Summary

- We have a function to optimize:

$$\log P(C|D, \lambda) = \sum_{(c,d) \in (C,D)} \log \frac{\exp \sum \lambda_i(c) f_i(d)}{\sum_c \exp \sum \lambda_i(c) f_i(d)}$$

- We know the function's derivatives:

$$\partial \log P(C|D, \lambda) / \partial \lambda_i(c) = \text{actual count}(f_i, c) - \text{predicted count}(f_i, \lambda)$$

- Ready to feed it into a numerical optimization package...
- What did any of that have to do with entropy?

## Smoothing: Issues of Scale

- Lots of features:
  - NLP maxent models can have over 1M features.
  - Even storing a single array of parameter values can have a substantial memory cost.
- Lots of sparsity:
  - Overfitting very easy – need smoothing!
  - Many features seen in training will never occur again at test time.
- Optimization problems:
  - Feature weights can be infinite, and iterative solvers can take a long time to get to those infinities.

## Smoothing: Issues

- Assume the following empirical distribution:

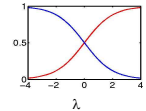
Heads	Tails
h	t

- Features: {Heads}, {Tails}
- We'll have the following model distribution:

$$P_{\text{HEADS}} = \frac{e^{\lambda h}}{e^{\lambda h} + e^{\lambda t}} \quad P_{\text{TAILS}} = \frac{e^{\lambda t}}{e^{\lambda h} + e^{\lambda t}}$$

- Really, only one degree of freedom ( $\lambda = \lambda_H - \lambda_T$ )

$$P_{\text{HEADS}} = \frac{e^{\lambda h} e^{-\lambda t}}{e^{\lambda h} e^{-\lambda t} + e^{\lambda t} e^{-\lambda t}} = \frac{e^{\lambda}}{e^{\lambda} + e^0} \quad P_{\text{TAILS}} = \frac{e^0}{e^{\lambda} + e^0}$$

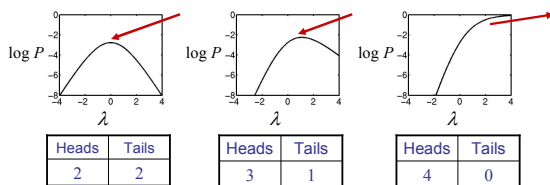


## Smoothing: Issues

- The data likelihood in this model is:

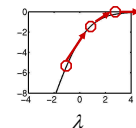
$$\log P(h, t | \lambda) = h \log p_{\text{HEADS}} + t \log p_{\text{TAILS}}$$

$$\log P(h, t | \lambda) = h\lambda - (t+h) \log(1 + e^{\lambda})$$



## Smoothing: Early Stopping

- In the 4/0 case, there were two problems:
  - The optimal value of  $\lambda$  was  $\infty$ , which is a long trip for an optimization procedure.
  - The learned distribution is just as spiked as the empirical one – no smoothing.



- One way to solve both issues is to just stop the optimization early, after a few iterations.

- The value of  $\lambda$  will be finite (but presumably big).
- The optimization won't take forever (clearly).
- Commonly used in early maxent work.

Heads	Tails
4	0

Heads	Tails
1	0

Output

## Smoothing: Priors (MAP)

- What if we had a prior expectation that parameter values wouldn't be very large?
- We could then balance evidence suggesting large parameters (or infinite) against our prior.
- The evidence would never totally defeat the prior, and parameters would be smoothed (and kept finite!).
- We can do this explicitly by changing the optimization objective to maximum posterior likelihood:

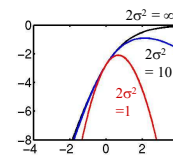
$$\log P(C, \lambda | D) = \log P(\lambda) + \log P(C | D, \lambda)$$

Posterior      Prior      Evidence

## Smoothing: Priors

- Gaussian, or quadratic, priors:
  - Intuition: parameters shouldn't be large.
  - Formalization: prior expectation that each parameter will be distributed according to a gaussian with mean  $\mu$  and variance  $\sigma^2$ .

$$P(\lambda_i) = \frac{1}{\sigma_i \sqrt{2\pi}} \exp\left(-\frac{(\lambda_i - \mu_i)^2}{2\sigma_i^2}\right)$$



They don't even capitalize my name anymore!



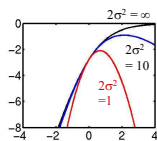
- Penalizes parameters for drifting too far from their mean prior value (usually  $\mu=0$ ).
- $2\sigma^2=1$  works surprisingly well (better to set using held-out data, though)

## Smoothing: Priors

- If we use gaussian priors:
  - Trade off some expectation-matching for smaller parameters.
  - When multiple features can be recruited to explain a data point, the more common ones generally receive more weight.
  - Accuracy generally goes up!

$$\log P(C, \lambda | D) = \log P(C | D, \lambda) - \log P(\lambda)$$

$$\log P(C, \lambda | D) = \sum_{(c,d) \in (C,D)} P(c | d, \lambda) - \sum_i \frac{(\lambda_i - \mu_i)^2}{2\sigma_i^2} + k$$



- Change the derivative:

$$\partial \log P(C, \lambda | D) / \partial \lambda_i = \text{actual}(f_i, C) - \text{predicted}(f_i, \lambda) - (\lambda_i - \mu_i) / \sigma^2$$

## Example: NER Smoothing

Because of smoothing, the more common prefixes have larger weights even though entire-word features are more specific.

Feature Weights

Feature Type	Feature	PERS	LOC
Previous word	at	-0.73	0.94
Current word	Grace	0.03	0.00
Beginning bigram	<G	0.45	-0.04
Current POS tag	NNP	0.47	0.45
Prev and cur tags	IN NNP	-0.10	0.14
Previous state	Other	-0.70	-0.92
Current signature	Xx	0.80	0.46
Prev state, cur sig	O-Xx	0.68	0.37
Prev-cur-next sig	x-Xx-Xx	-0.69	0.37
P. state - p-cur sig	O-x-Xx	-0.20	0.82
...			
<b>Total:</b>		<b>-0.58</b>	<b>2.68</b>

Local Context

	Prev	Cur	Next
State	Other	???	???
Word	at	Grace	Road
Tag	IN	NNP	NNP
Sig	x	Xx	Xx