Midterm Review

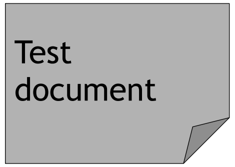
Instructor: Wei Xu


HAPPY MIDTERM WEEK
AND MAY THE ODDS BE EVER IN YOUR FAVOR
LIONSGATE
memecrunch.com

Classification

Naïve Bayes, Logistic Regression (Log-linear Models), Perceptron, Gradient Descent

Text Classification



parser
language
label
translation
…

?

Machine Learning	NLP	Garbage Collection	Planning	GUI
learning	parser	garbage	planning	...
training	tag	collection	temporal	
algorithm	training	memory	reasoning	
shrinkage	translation	optimization	plan	
network...	language...	region...	language...	

Classification Methods: Supervised Machine Learning

- *Input:*
  - a document d
  - a fixed set of classes $C = \{c_1, c_2, \ldots, c_J\}$
  - A training set of m hand-labeled documents $(d_1,c_1),\ldots,(d_m,c_m)$
- *Output:*
  - a learned classifier $y{:}d \rightarrow c$

Naïve Bayes Classifier

$$c_{MAP} = \underset{c \in C}{\operatorname{argmax}} P(c \mid d)$$

MAP is "maximum a posteriori" = most likely class

$$= \underset{c \in C}{\operatorname{argmax}} \frac{P(d \mid c) P(c)}{P(d)}$$

Bayes Rule

$$= \underset{c \in C}{\operatorname{argmax}} P(d \mid c) P(c)$$

Dropping the denominator

$$= \underset{c \in C}{\operatorname{argmax}} P(x_1, x_2, \ldots, x_n \mid c) P(c)$$

bag of word

$$c_{NB} = \underset{c \in C}{\operatorname{argmax}} P(c_j) \prod_{x \in X} P(x \mid c)$$

conditional independence assumption

Multinomial Naïve Bayes: Learning

- maximum likelihood estimates
  - simply use the frequencies in the data

$$\hat{P}(c_j) = \frac{doccount(C = c_j)}{N_{doc}}$$

$$\hat{P}(w_i \mid c_j) = \frac{count(w_i, c) + 1}{\left(\sum_{w \in V} count(w, c)\right) + |V|}$$

laplace (add-1) smoothing to avoid zero probabilities

- For *unknown* words (which completely doesn't occur in training set), we can ignore them.

Weaknesses of Naïve Bayes

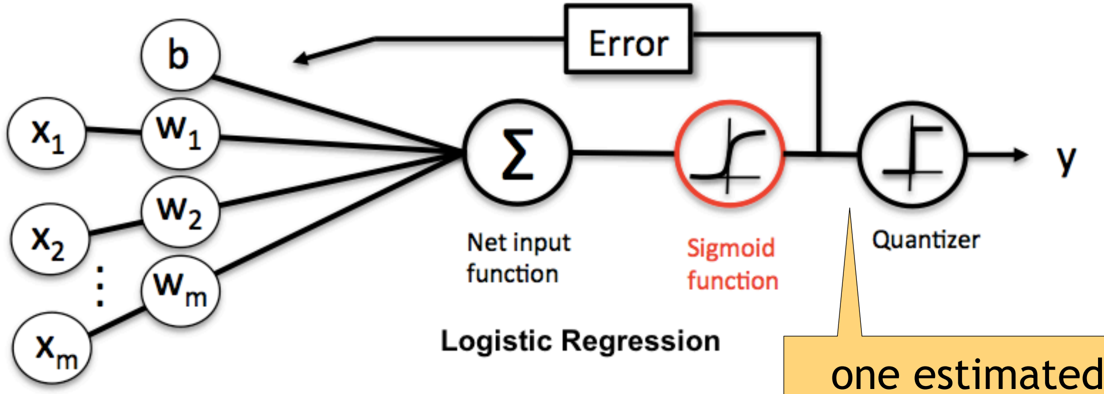
- Assuming conditional independence
- Correlated features -> double counting evidence
  - Parameters are estimated independently
- This can hurt classifier accuracy and calibration

Logistic Regression

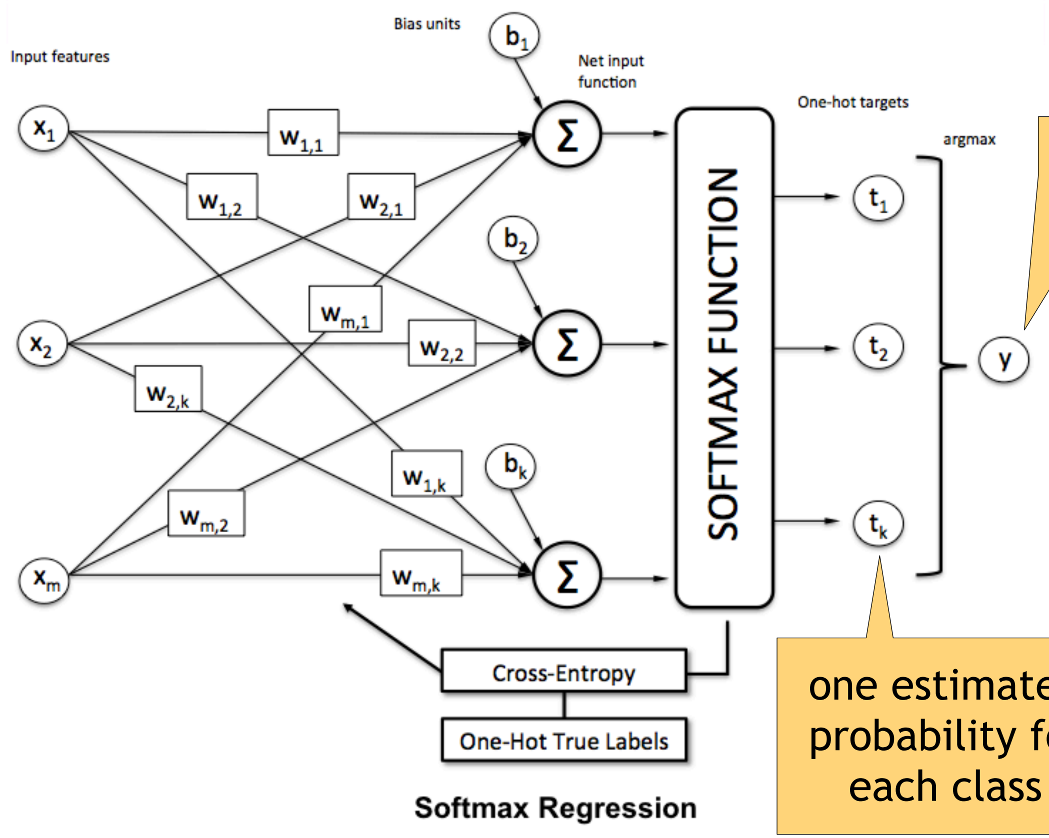
- Doesn't assume conditional independence of features
  - Better calibrated probabilities
  - Can handle highly correlated overlapping features

Logistic vs. Softmax Regression

Generalization
to Multi-class Problems



one estimated probability
good for binary classification



the class with
highest
probability

one estimated
probability for
each class

Maximum Entropy Models (MaxEnt)

- a.k.a **logistic regression** or **multinominal logistic regression** or **multiclass logistic regression** or **softmax regression**

- belongs to the family of classifiers know as log-linear classifiers

- Math proof of “LR=MaxEnt”:
  - [Klein and Manning 2003]
  - [Mount 2011]

http://www.win-vector.com/dfiles/LogisticRegressionMaxEnt.pdf

Log-Linear Models

- We have some input domain $\mathcal{X}$, and a finite label set $\mathcal{Y}$. Aim is to provide a conditional probability $p(y \mid x)$ for any $x \in \mathcal{X}$ and $y \in \mathcal{Y}$.
- A feature is a function $f : \mathcal{X} \times \mathcal{Y} \rightarrow \mathbb{R}$
(Often binary features or indicator functions
$f_k : \mathcal{X} \times \mathcal{Y} \rightarrow \{0, 1\}$).
- Say we have m features f_k for $k = 1 \ldots m$
$\Rightarrow$ A feature vector $f(x, y) \in \mathbb{R}^m$ for any $x \in \mathcal{X}$ and $y \in \mathcal{Y}$.
- We also have a **parameter vector** $v \in \mathbb{R}^m$
- We define

softmax function

$$p(y \mid x; v) = \frac{e^{v \cdot f(x,y)}}{\sum_{y' \in \mathcal{Y}} e^{v \cdot f(x,y')}}$$

convert into probabilities between [0, 1]

Maximum-Likelihood Estimation

- Maximum-likelihood estimates given training sample $(x^{(i)}, y^{(i)})$ for $i = 1 \ldots n$, each $(x^{(i)}, y^{(i)}) \in \mathcal{X} \times \mathcal{Y}$:

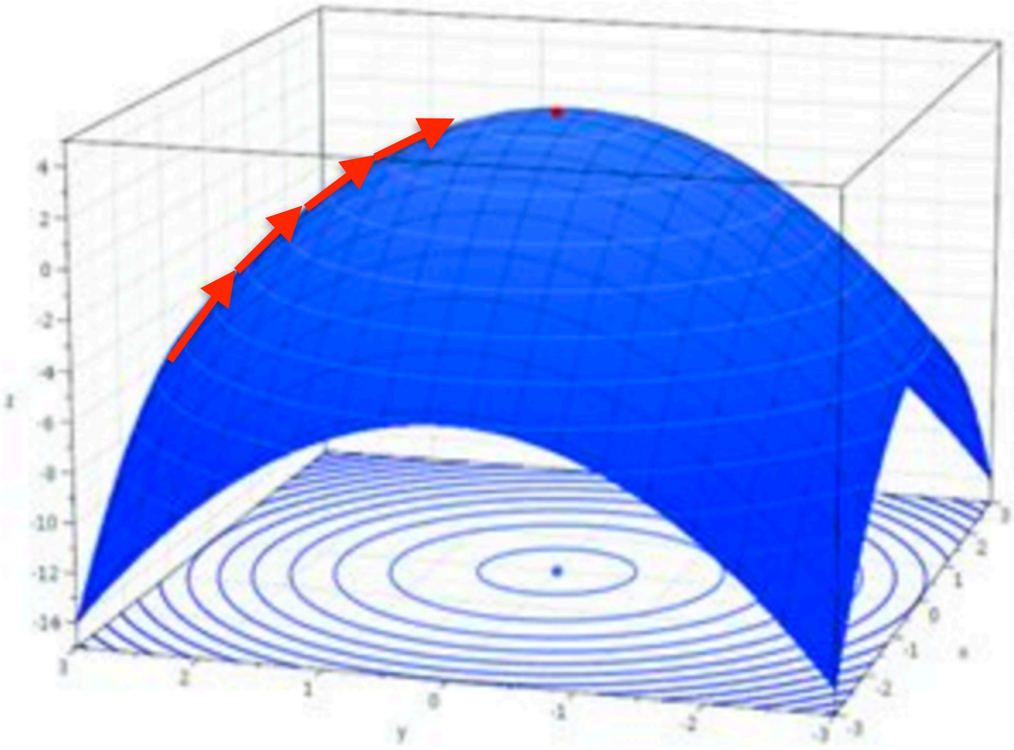
$$v_{ML} = \text{argmax}_{v \in \mathbb{R}^m} L(v)$$

where

$$L(v) = \sum_{i=1}^{n} \log p(y^{(i)} \mid x^{(i)}; v) = \sum_{i=1}^{n} v \cdot f(x^{(i)}, y^{(i)}) - \sum_{i=1}^{n} \log \sum_{y' \in \mathcal{Y}} e^{v \cdot f(x^{(i)}, y')}$$

concave function!

Gradient Ascent



Gradient Ascent

Loop While not converged:

For all features **j**, compute and add derivatives

$$w_j^{\text{new}} = w_j^{\text{old}} + \eta \frac{\partial}{\partial w_j} \mathcal{L}(w)$$

$\mathcal{L}(w)$: Training set log-likelihood (Objective function)

$\left(\frac{\partial \mathcal{L}}{\partial w_1}, \frac{\partial \mathcal{L}}{\partial w_2}, \ldots, \frac{\partial \mathcal{L}}{\partial w_n} \right)$: Gradient vector

Smoothing in Log-Linear Models

- Say we have a feature:

$$f_{100}(x, y) = \begin{cases} 1 & \text{if current word } w_i \text{ is base and } y = \texttt{Vt} \\ 0 & \text{otherwise} \end{cases}$$

- In training data, base is seen 3 times, with Vt every time
- Maximum likelihood solution satisfies

$$\sum_i f_{100}(x^{(i)}, y^{(i)}) = \sum_i \sum_y p(y \mid x^{(i)}; v) f_{100}(x^{(i)}, y)$$

$\Rightarrow p(\text{Vt} \mid x^{(i)}; v) = 1$ for any history $x^{(i)}$ where $w_i =$ base
$\Rightarrow v_{100} \rightarrow \infty$ at maximum-likelihood solution (most likely)
$\Rightarrow p(\text{Vt} \mid x; v) = 1$ for any test data history x where $w =$ base

Regularization

- Modified loss function

$$L(v) = \sum_{i=1}^{n} v \cdot f(x^{(i)}, y^{(i)}) - \sum_{i=1}^{n} \log \sum_{y' \in \mathcal{Y}} e^{v \cdot f(x^{(i)}, y')} - \frac{\lambda}{2} \sum_{k=1}^{m} v_k^2$$

- Calculating gradients:

$$\frac{dL(v)}{dv_k} = \underbrace{\sum_{i=1}^{n} f_k(x^{(i)}, y^{(i)})}_{\text{Empirical counts}} - \underbrace{\sum_{i=1}^{n} \sum_{y' \in \mathcal{Y}} f_k(x^{(i)}, y') p(y' \mid x^{(i)}; v)}_{\text{Expected counts}} - \lambda v_k$$

- Can run conjugate gradient methods as before
- Adds a penalty for large weights

LR Gradient

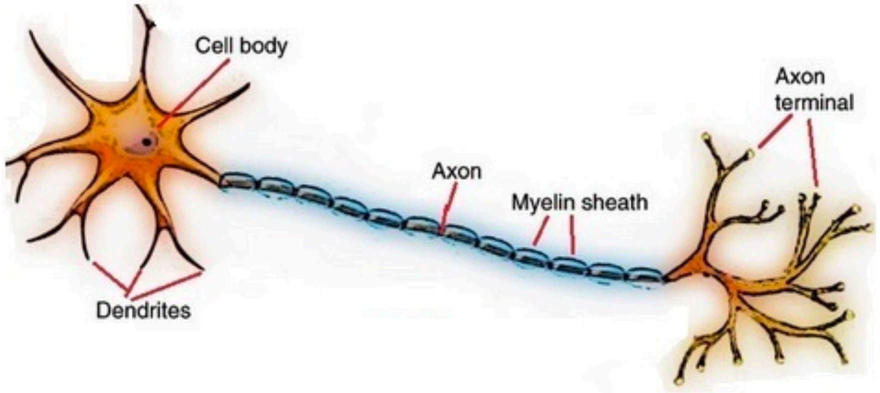
$$w_{\text{MLE}} = \text{argmax}_w \underbrace{\sum_i y_i \log p_i + (1 - y_i) \log(1 - p_i)}_{\mathcal{L}}$$

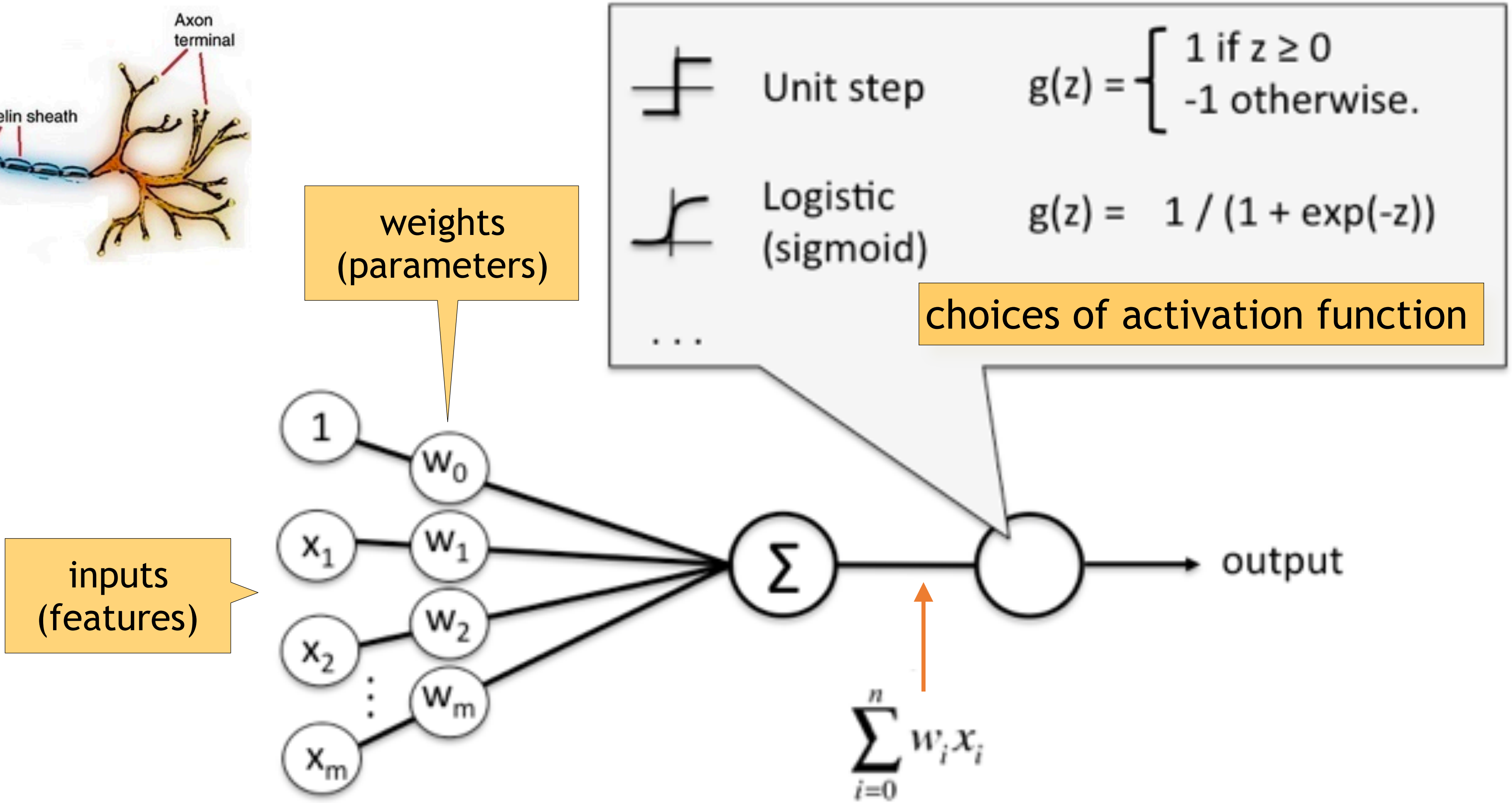
logistic function

$$\boxed{p_i = \sigma(\textstyle\sum_j w_j x_j)}$$

$$\frac{\partial \mathcal{L}}{\partial w_j} = \sum_i (y_i - p_i) x_j$$

Perceptron vs. Logistic Regression



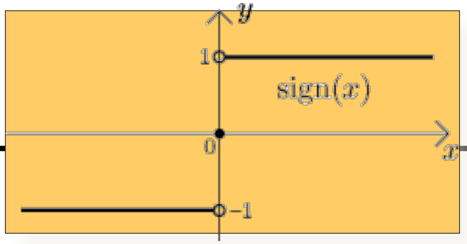


Perceptron Algorithm

- Very similar to logistic regression
- Not exactly computing gradient

```
Initialize weight vector w = 0
Loop for K iterations
    Loop For all training examples x_i
        if sign(w * x_i) != y_i
            w += y_i * x_i
```

Error-driven!



MultiClass Perceptron Algorithm

```
Initialize weight vector w = 0
Loop for K iterations
    Loop For all training examples x_i
        y_pred = argmax_y w_y * x_i
        if y_pred != y_i
            w_y_gold += x_i
            w_y_pred -= x_i
```

increase score for right answer

decrease score for wrong answer

Perceptron vs. Logistic Regression

- Only hyperparameter of perceptron is maximum number of iterations (LR also needs learning rate)

- Perceptron is guaranteed to converge if the data is linearly separable (LR always converge)

Perceptron vs. Logistic Regression

- The Perceptron is an online learning algorithm.
- Logistic Regression is not:

this update is effectively the same as "w += y_i * x_i"

$$w_{\mathrm{MLE}} = \mathrm{argmax}_w \log P(y_1, \ldots, y_d | x_1, \ldots, x_d; w)$$

$$= \mathrm{argmax}_w \sum_i y_i \log p_i + (1 - y_i) \log(1 - p_i)$$

Multinominal LR Gradient

$$\frac{\partial \mathcal{L}}{\partial w_j} = \sum_{i=1}^{D} f_j(y_i, d_i) - \sum_{i=1}^{D} \sum_{y \in Y} f_j(y, d_i) P(y|d_i)$$

empirical feature count

expected feature count

MAP-based learning (Perceptron)

Maximum A Posteriori

$$\frac{\partial \mathcal{L}}{\partial w_j} \approx \sum_{i=1}^{D} f_j(y_i, d_i) - \sum_{i=1}^{D} f_j(\arg\max_{y \in Y} P(y|d_i), d_i)$$

approximate using maximization

Language Modeling

Markov Assumption, Perplexity, Interpolation, Backoff, and Kneser-Ney Smoothing

Probabilistic Language Modeling

- Goal: compute the probability of a sentence or sequence of words:

$$P(W) = P(w_1,w_2,w_3,w_4,w_5 \ldots w_n)$$

- Related task: probability of an upcoming word:

$$P(w_5|w_1,w_2,w_3,w_4)$$

- A model that computes either of these:

$P(W)$ or $P(w_n|w_1,w_2 \ldots w_{n-1})$ is called a **language model** or **LM**

The Chain Rule applied to compute joint probability of words in sentence

$$P(w_1 w_2 \ldots w_n) = \prod_i P(w_i \mid w_1 w_2 \ldots w_{i-1})$$

Markov Assumption

$$P(w_i \mid w_1 w_2 \ldots w_{i-1}) \approx P(w_i \mid w_{i-k} \ldots w_{i-1})$$

Bigram model

- Condition on the previous word:

$$P(w_i \mid w_1 w_2 \ldots w_{i-1}) \approx P(w_i \mid w_{i-1})$$

Add-one estimation

- Also called Laplace smoothing
- Pretend we saw each word one more time than we did
- Just add one to all the counts!

- MLE estimate:

$$P_{MLE}(w_i \mid w_{i-1}) = \frac{c(w_{i-1}, w_i)}{c(w_{i-1})}$$

- Add-1 estimate:

$$P_{Add-1}(w_i \mid w_{i-1}) = \frac{c(w_{i-1}, w_i) + 1}{c(w_{i-1}) + V}$$

Add-1 estimation is a blunt instrument

- So add-1 isn’t used for N-grams:
  - We’ll see better methods

- But add-1 is used to smooth other NLP models
  - For text classification
  - In domains where the number of zeros isn’t so huge.

Better Language Models

- Linear interpolation

$$\begin{aligned}\hat{P}(w_n|w_{n-2}w_{n-1}) \;=\; & \lambda_1 P(w_n|w_{n-2}w_{n-1}) \\ & +\lambda_2 P(w_n|w_{n-1}) \\ & +\lambda_3 P(w_n)\end{aligned}$$

$$\sum_i \lambda_i = 1$$

- Backoff
- Absolute Discounting
- Kneser-Ney Smoothing

Perplexity

The best language model is one that best predicts an unseen test set

- Gives the highest P(sentence)

$$PP(W) = P(w_1 w_2 \ldots w_N)^{-\frac{1}{N}}$$

Perplexity is the inverse probability of the test set, "normalized" by the number of words:

$$= \sqrt[N]{\frac{1}{P(w_1 w_2 \ldots w_N)}}$$

Chain Rule

$$\mathrm{PP}(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i | w_1 \ldots w_{i-1})}}$$

for bigram

$$\mathrm{PP}(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i | w_{i-1})}}$$

Minimizing perplexity is the same as maximizing probability

Tagging

Hidden Markov Models, Maximum Entropy Markov Models (Log-linear Models for Tagging), and Viterbi Algorithm

Tagging (Sequence Labeling)

- Given a sequence (in NLP, words), assign appropriate labels to each word.
- Many NLP problems can be viewed as sequence labeling:
  - POS Tagging
  - Chunking
  - Named Entity Tagging
- Labels of tokens are dependent on the labels of other tokens in the sequence, particularly their neighbors

Plays	well	with	others.
VBZ	RB	IN	NNS

Two Types of Constraints

Influential/JJ members/NNS of/IN the/DT House/NNP Ways/NNP and/CC Means/NNP Committee/NNP introduced/VBD legislation/NN that/WDT would/MD restrict/VB how/WRB the/DT new/JJ savings-and-loan/NN bailout/NN agency/NN can/MD raise/VB capital/NN ./.

- "Local": e.g., *can* is more likely to be a modal verb MD rather than a noun NN
- "Contextual": e.g., a noun is much more likely than a verb to follow a determiner
- Sometimes these preferences are in conflict:

  The trash can is in the garage

Hidden Markov Models

- We have an input sentence $x = x_1, x_2, \ldots, x_n$ (x_i is the i'th word in the sentence)
- We have a tag sequence $y = y_1, y_2, \ldots, y_n$ (y_i is the i'th tag in the sentence)
- We'll use an HMM to define

$$p(x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_n)$$

  for any sentence $x_1 \ldots x_n$ and tag sequence $y_1 \ldots y_n$ of the same length.
- Then the most likely tag sequence for x is

$$\arg\max_{y_1 \cdots y_n} p(x_1 \ldots x_n, y_1, y_2, \ldots, y_n)$$

Trigram Hidden Markov Models (Trigram HMMs)

For any sentence $x_1 \dots x_n$ where $x_i \in \mathcal{V}$ for $i = 1 \dots n$, and any tag sequence $y_1 \dots y_{n+1}$ where $y_i \in \mathcal{S}$ for $i = 1 \dots n$, and $y_{n+1} =$ STOP, the joint probability of the sentence and tag sequence is

$$p(x_1 \dots x_n, y_1 \dots y_{n+1}) = \prod_{i=1}^{n+1} q(y_i | y_{i-2}, y_{i-1}) \prod_{i=1}^{n} e(x_i | y_i)$$

where we have assumed that $x_0 = x_{-1} = *$.

Parameters of the model:

- $q(s|u, v)$ for any $s \in \mathcal{S} \cup \{\text{STOP}\}$, $u, v \in \mathcal{S} \cup \{*\}$ Trigram parameters
- $e(x|s)$ for any $s \in \mathcal{S}$, $x \in \mathcal{V}$ Emission parameters

An Example

If we have $n = 3$, $x_1 \ldots x_3$ equal to the sentence *the dog laughs*, and $y_1 \ldots y_4$ equal to the tag sequence `D N V STOP`, then

$$
\begin{aligned}
& p(x_1 \ldots x_n, y_1 \ldots y_{n+1}) \\
= \; & q(\texttt{D}|*,*) \times q(\texttt{N}|*,\texttt{D}) \times q(\texttt{V}|\texttt{D},\texttt{N}) \times q(\texttt{STOP}|\texttt{N},\texttt{V}) \\
& \times e(\textit{the}|\texttt{D}) \times e(\textit{dog}|\texttt{N}) \times e(\textit{laughs}|\texttt{V})
\end{aligned}
$$

- STOP is a special tag that terminates the sequence
- We take $y_0 = y_{-1} = *$, where * is a special "padding" symbol

The Viterbi Algorithm

Problem: for an input $x_1 \ldots x_n$, find

$$\arg \max_{y_1 \cdots y_{n+1}} p(x_1 \ldots x_n, y_1 \ldots y_{n+1})$$

where the $\arg\max$ is taken over all sequences $y_1 \ldots y_{n+1}$ such that $y_i \in \mathcal{S}$ for $i = 1 \ldots n$, and $y_{n+1} =$ STOP.

We assume that p again takes the form

$$p(x_1 \ldots x_n, y_1 \ldots y_{n+1}) = \prod_{i=1}^{n+1} q(y_i | y_{i-2}, y_{i-1}) \prod_{i=1}^{n} e(x_i | y_i)$$

Recall that we have assumed in this definition that $y_0 = y_{-1} = *$, and $y_{n+1} =$ STOP.

Brute Force Search is Hopelessly Inefficient

Problem: for an input $x_1 \ldots x_n$, find

$$\arg \max_{y_1 \cdots y_{n+1}} p(x_1 \ldots x_n, y_1 \ldots y_{n+1})$$

where the $\arg\max$ is taken over all sequences $y_1 \ldots y_{n+1}$ such that $y_i \in \mathcal{S}$ for $i = 1 \ldots n$, and $y_{n+1} = \text{STOP}$.

The Viterbi Algorithm

Andrew Viterbi, 1967

- Define n to be the length of the sentence
- Define S_k for $k = -1 \ldots n$ to be the set of possible tags at position k:

$$S_{-1} = S_0 = \{*\}$$
$$S_k = S \quad \text{for } k \in \{1 \ldots n\}$$

- Define

$$r(y_{-1}, y_0, y_1, \ldots, y_k) = \prod_{i=1}^{k} q(y_i|y_{i-2}, y_{i-1}) \prod_{i=1}^{k} e(x_i|y_i)$$

- Define a dynamic programming table

$$\pi(k, u, v) = \text{maximum probability of a tag sequence ending in tags } u, v \text{ at position } k$$

that is,
$\pi(k, u, v) = \max_{\langle y_{-1}, y_0, y_1, \ldots, y_k \rangle : y_{k-1}=u, y_k=v} r(y_{-1}, y_0, y_1 \ldots y_k)$

A Recursive Definition

Base case:

$$\pi(0, *, *) = 1$$

Recursive definition:
For any $k \in \{1 \ldots n\}$, for any $u \in \mathcal{S}_{k-1}$ and $v \in \mathcal{S}_k$:

$$\pi(k, u, v) = \max_{w \in \mathcal{S}_{k-2}} \left(\pi(k-1, w, u) \times q(v|w, u) \times e(x_k|v)\right)$$

The Viterbi Algorithm

Input: a sentence $x_1 \ldots x_n$, parameters $q(s|u,v)$ and $e(x|s)$.

Initialization: Set $\pi(0, *, *) = 1$

Definition: $\mathcal{S}_{-1} = \mathcal{S}_0 = \{*\}$, $\mathcal{S}_k = \mathcal{S}$ for $k \in \{1 \ldots n\}$

Algorithm:

- For $k = 1 \ldots n$,
  - For $u \in \mathcal{S}_{k-1}$, $v \in \mathcal{S}_k$,

$$\pi(k,u,v) = \max_{w \in \mathcal{S}_{k-2}} \left(\pi(k-1,w,u) \times q(v|w,u) \times e(x_k|v)\right)$$

- **Return** $\max_{u \in \mathcal{S}_{n-1}, v \in \mathcal{S}_n} \left(\pi(n,u,v) \times q(\mathsf{STOP}|u,v)\right)$

The Viterbi Algorithm: Running Time

running time for trigram HMM

running time for bigram HMM is O(n|S|²)

- $O(n|\mathcal{S}|^3)$ time to calculate $q(s|u,v) \times e(x_k|s)$ for all k, s, u, v.
- $n|\mathcal{S}|^2$ entries in π to be filled in.
- $O(|\mathcal{S}|)$ time to fill in one entry
- $\Rightarrow O(n|\mathcal{S}|^3)$ time in total

Pros and Cons

- Hidden markov model taggers are very simple to train (just need to compile counts from the training corpus)
- Perform relatively well (over 90% performance on named entity recognition)
- Main difficulty is modeling

$$e(word \mid tag)$$

can be very difficult if “words” are complex

Log-Linear Models for Tagging

- We have an input sentence $w_{[1:n]} = w_1, w_2, \ldots, w_n$ (w_i is the i'th word in the sentence)
- We have a tag sequence $t_{[1:n]} = t_1, t_2, \ldots, t_n$ (t_i is the i'th tag in the sentence)
- We'll use an log-linear model to define

$$p(t_1, t_2, \ldots, t_n | w_1, w_2, \ldots, w_n)$$

  for any sentence $w_{[1:n]}$ and tag sequence $t_{[1:n]}$ of the same length.
  (Note: contrast with HMM that defines $p(t_1 \ldots t_n, w_1 \ldots w_n)$)
- Then the most likely tag sequence for $w_{[1:n]}$ is

$$t^*_{[1:n]} = \operatorname{argmax}_{t_{[1:n]}} p(t_{[1:n]} | w_{[1:n]})$$

How to model $p(t_{[1:n]}|w_{[1:n]})$?

A Trigram Log-Linear Tagger:

$$p(t_{[1:n]}|w_{[1:n]}) = \prod_{j=1}^{n} p(t_j \mid w_1 \dots w_n, t_1 \dots t_{j-1})$$ Chain rule

$$= \prod_{j=1}^{n} p(t_j \mid w_1, \dots, w_n, t_{j-2}, t_{j-1})$$

Independence assumptions

- We take $t_0 = t_{-1} = *$
- Independence assumption: each tag only depends on previous two tags

$$p(t_j|w_1, \dots, w_n, t_1, \dots, t_{j-1}) = p(t_j|w_1, \dots, w_n, t_{j-2}, t_{j-1})$$

Decoding

- **Linear Perceptron** $s^* = \arg\max_s w \cdot \Phi(x, s)$
  - Features must be local, for $x=x_1 \ldots x_m$, and $s=s_1 \ldots s_m$

$$\Phi(x, s) = \sum_{j=1}^{m} \phi(x, j, s_{j-1}, s_j)$$

  - Define $\pi(i,s_i)$ to be the max score of a sequence of length i ending in tag s_i

$$\pi(i, s_i) = \max_{s_{i-1}} w \cdot \phi(x, i, s_{i-i}, s_i) + \pi(i-1, s_{i-1})$$

- **Viterbi algorithm (HMMs):**

$$\pi(i, s_i) = \max_{s_{i-1}} e(x_i|s_i) q(s_i|s_{i-1}) \pi(i-1, s_{i-1})$$

- **Viterbi algorithm (Maxent):**

$$\pi(i, s_i) = \max_{s_{i-1}} p(s_i|s_{i-1}, x_1 \ldots x_m) \pi(i-1, s_{i-1})$$

Summary

- Key ideas in log-linear taggers:
  - Decompose
    $$p(t_1 \dots t_n | w_1 \dots w_n) = \prod_{i=1}^{n} p(t_i | t_{i-2}, t_{i-1}, w_1 \dots w_n)$$
  - Estimate
    $$p(t_i | t_{i-2}, t_{i-1}, w_1 \dots w_n)$$
    using a log-linear model
  - For a test sentence $w_1 \dots w_n$, use the Viterbi algorithm to find
    $$\arg\max_{t_1 \dots t_n} \left(\prod_{i=1}^{n} p(t_i | t_{i-2}, t_{i-1}, w_1 \dots w_n) \right)$$
- Key advantage over HMM taggers: flexibility in the features they can use