



# Lecture 9 – Sequence Modelling with Features: Linear-Chain Conditional Random Fields

**Instructor:** Jackie CK Cheung  
COMP-550

Sutton and McCallum:  
<http://homepages.inf.ed.ac.uk/csutton/publications/crftutv2.pdf> Sections 1, 2–2.3, 3–3.1;

Eisenstein, Section 7.5

# Outline

---

Hidden Markov models: shortcomings

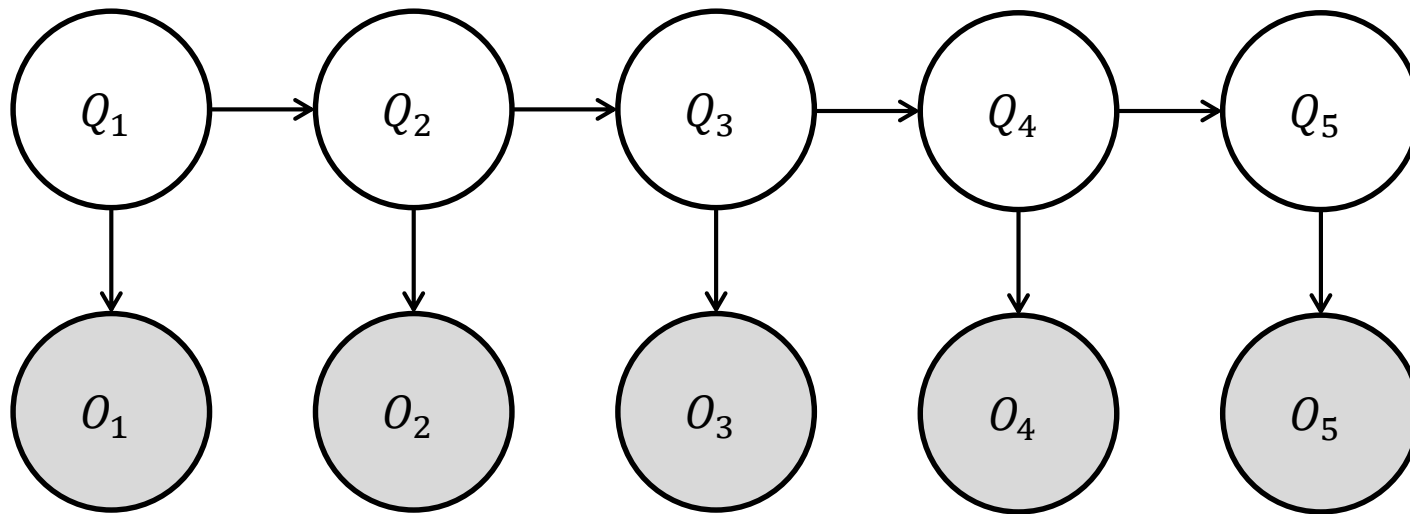
Generative vs. discriminative models

Linear-chain CRFs

- Inference and learning algorithms with linear-chain CRFs

# Hidden Markov Model

Graph specifies how joint probability decomposes



$$P(\mathbf{O}, \mathbf{Q}) = \underbrace{P(Q_1)}_{\text{Initial state probability}} \prod_{t=1}^{T-1} \underbrace{P(Q_{t+1}|Q_t)}_{\text{State transition probabilities}} \prod_{t=1}^T \underbrace{P(O_t|Q_t)}_{\text{Emission probabilities}}$$

Initial state probability

State transition probabilities

Emission probabilities

# Shortcomings of Standard HMMs

---

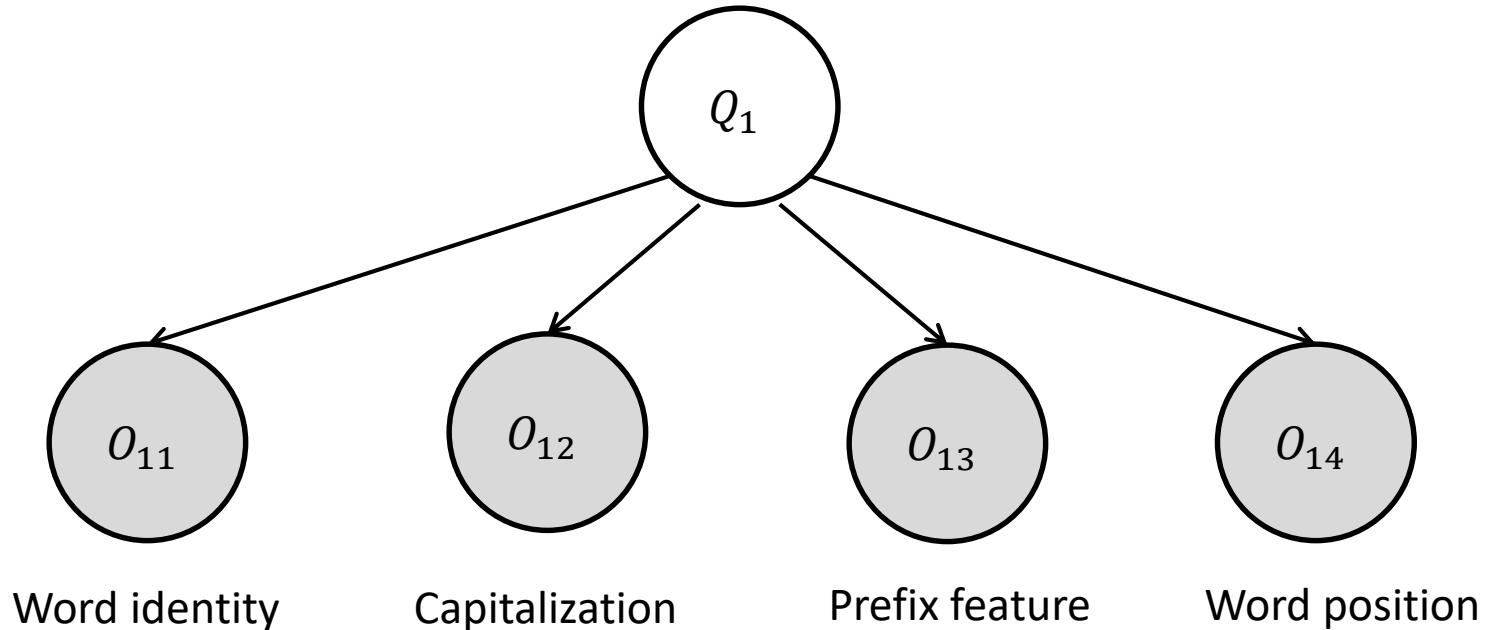
How do we add more features to HMMs?

Might be useful for POS tagging:

- Word position within sentence (1<sup>st</sup>, 2<sup>nd</sup>, last...)
- Capitalization
- Word prefix and suffixes (*-tion, -ed, -ly, -er, re-, de-*)
- Features that depend on more than the current word or the previous words.

# Possible to Do with HMMs

Add more emissions at each timestep



Clunky

Is there a better way to do this?

# Discriminative Models

---

HMMs are **generative models**

- Build a model of the joint probability distribution  $P(\mathbf{O}, \mathbf{Q})$ ,
- Let's rename the variables
- Generative models specify  $P(X, Y; \theta^{\text{gen}})$

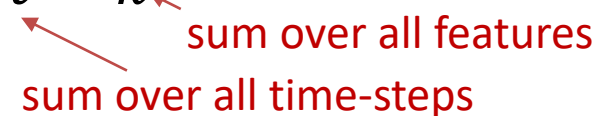
If we are only interested in POS tagging, we can instead train a **discriminative model**

- Model specifies  $P(Y|X; \theta^{\text{disc}})$
- Now a task-specific model for sequence labelling; cannot use it for generating new samples of word and POS sequences

# Discriminative Sequence Model


The parallel to an HMM in the discriminative case:  
**linear-chain conditional random fields (linear-chain CRFs)** (Lafferty et al., 2001)

$$P(Y|X) = \frac{1}{Z(X)} \exp \sum_t \sum_k \theta_k f_k(y_t, y_{t-1}, x_t)$$



$Z(X)$  is a normalization constant:

$$Z(X) = \sum_y \exp \sum_t \sum_k \theta_k f_k(y_t, y_{t-1}, x_t)$$



# Intuition

---

**Standard HMM:** product of probabilities; these probabilities are defined over the identity of the states and words

- Transition from state DT to NN:  $P(y_{t+1} = NN | y_t = DT)$
- Emit word the from state DT:  $P(x_t = the | y_t = DT)$

**Linear-chain CRF:** replace the products by numbers that are NOT probabilities, but linear combinations of weights and feature values.



# Features in CRFs

---

Standard HMM probabilities as CRF features:

- Transition from state DT to state NN

$$f_{DT \rightarrow NN}(y_t, y_{t-1}, x_t) = \mathbf{1}(y_{t-1} = DT) \mathbf{1}(y_t = NN)$$

- Emit *the* from state DT

$$f_{DT \rightarrow the}(y_t, y_{t-1}, x_t) = \mathbf{1}(y_t = DT) \mathbf{1}(x_t = the)$$

- Initial state is DT

$$f_{DT}(y_1, x_1) = \mathbf{1}(y_1 = DT)$$

**Indicator function:**

$$\text{Let } \mathbf{1}(\textit{condition}) = \begin{cases} 1 & \text{if } \textit{condition} \text{ is true} \\ 0 & \text{otherwise} \end{cases}$$

# Features in CRFs

---

More examples of features that may be useful

- Word is capitalized

$$f_{cap}(y_t, y_{t-1}, x_t) = \mathbf{1}(y_t = ?) \mathbf{1}(x_t \text{ is capitalized})$$

- Word ends in *-ed*

$$f_{-ed}(y_t, y_{t-1}, x_t) = \mathbf{1}(y_t = ?) \mathbf{1}(x_t \text{ ends with } ed)$$

# Inference with LC-CRFs

Dynamic programming still works – modify the forward and the Viterbi algorithms to work with the weight-feature products.

	HMM	LC-CRF
Forward algorithm	$P(X \theta)$	$Z(X)$
Viterbi algorithm	$\operatorname{argmax}_Y P(X, Y \theta)$	$\operatorname{argmax}_Y P(Y X, \theta)$

# Forward Algorithm for HMMs

---

Create trellis  $\alpha_i(t)$  for  $i = 1 \dots N, t = 1 \dots T$

$\alpha_j(1) = \pi_j b_j(O_1)$  for  $j = 1 \dots N$

for  $t = 2 \dots T$ :

for  $j = 1 \dots N$ :

$$\alpha_j(t) = \sum_{i=1}^N \alpha_i(t-1) a_{ij} b_j(O_t)$$

$$P(\mathbf{O}|\theta) = \sum_{j=1}^N \alpha_j(T)$$

Runtime:  $O(N^2 T)$

# Forward Algorithm for LC-CRFs

Create trellis  $\alpha_i(t)$  for  $i = 1 \dots N, t = 1 \dots T$

$$\alpha_j(1) = \exp \sum_k \theta_k^{init} f_k^{init}(y_1 = j, x_1) \text{ for } j = 1 \dots N$$

for  $t = 2 \dots T$ :

for  $j = 1 \dots N$ :

$$\alpha_j(t) = \sum_{i=1}^N \alpha_i(t-1) \exp \sum_k \theta_k f_k(y_t = j, y_{t-1}, x_t)$$

$$Z(X) = \sum_{j=1}^N \alpha_j(T)$$

Transition and emission probabilities replaced  
by exponent of weighted sums of features.

Runtime:  $O(N^2 T)$

Having  $Z(X)$  allows us to compute  $P(Y|X)$

# Viterbi Algorithm for HMMs

---

Create trellis  $\delta_i(t)$  for  $i = 1 \dots N, t = 1 \dots T$

$\delta_j(1) = \pi_j b_j(O_1)$  for  $j = 1 \dots N$

for  $t = 2 \dots T$ :

    for  $j = 1 \dots N$ :

$$\delta_j(t) = \max_i \delta_i(t-1) a_{ij} b_j(O_t)$$

Take  $\max_i \delta_i(T)$

Runtime:  $O(N^2 T)$

# Viterbi Algorithm for LC-CRFs

---

Create trellis  $\delta_i(t)$  for  $i = 1 \dots N, t = 1 \dots T$

$$\delta_j(1) = \exp \sum_k \theta_k^{init} f_k^{init}(y_1 = j, x_1) \text{ for } j = 1 \dots N$$

for  $t = 2 \dots T$ :

for  $j = 1 \dots N$ :

$$\delta_j(t) = \max_i \delta_i(t-1) \exp \sum_k \theta_k f_k(y_t = j, y_{t-1}, x_t)$$

Take  $\max_i \delta_i(T)$

Runtime:  $O(N^2 T)$

Remember that we need backpointers.

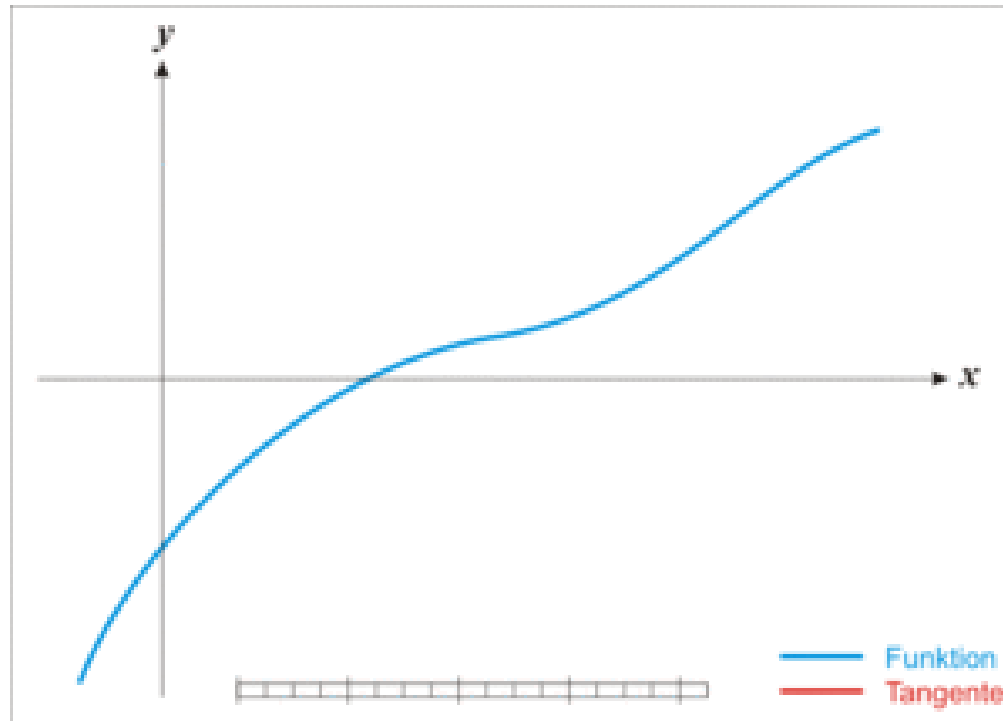
# Training LC-CRFs

Unlike for HMMs, no analytic MLE solution

Use iterative method to improve data likelihood

## Gradient descent

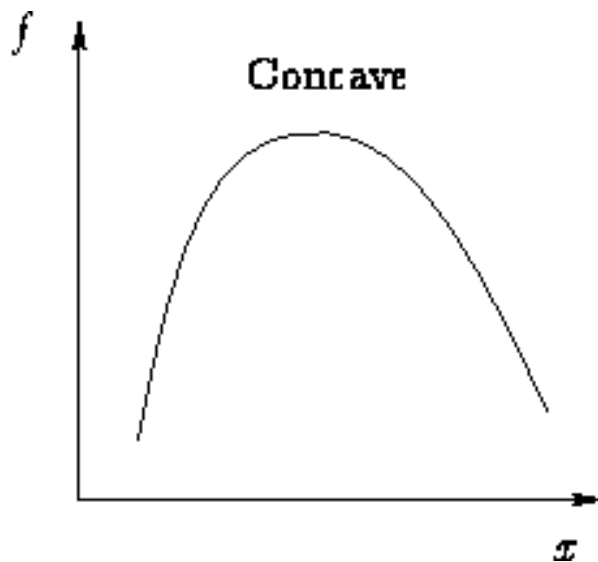
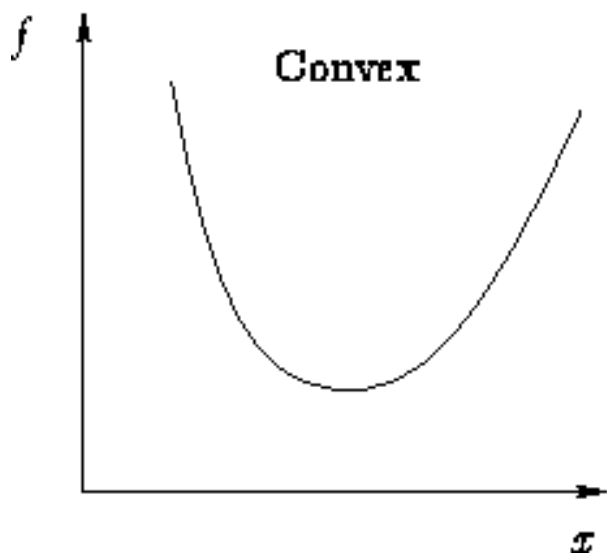
A version of Newton's method to find where the gradient is 0





# Convexity

Fortunately,  $l(\theta)$  is a **concave** function (equivalently, its negation is a **convex** function). That means that we will find the global maximum of  $l(\theta)$  with gradient ascent (equivalently, the global minimum of  $-l(\theta)$  with gradient descent).



# Gradient Ascent

---

Walk in the direction of the gradient to maximize  $l(\theta)$

- a.k.a., gradient descent on a loss function

$$\theta^{\text{new}} = \theta^{\text{old}} + \gamma \nabla l(\theta)$$

$\gamma$  is a learning rate that specifies how large a step to take.

There are more sophisticated ways to do this update:

- Conjugate gradient
- L-BFGS (approximates using second derivative)

# Gradient Descent Summary

---

## Descent vs ascent

Convention: think about the problem as a minimization problem

*Minimize the negative log likelihood*

- $\theta \leftarrow \theta - \gamma(-\nabla l(\theta))$

Initialize  $\theta = \{\theta_1, \theta_2, \dots, \theta_k\}$  randomly

Do for a while:

Compute  $\nabla l(\theta)$ , which will require dynamic programming  
(i.e., forward algorithm)

$$\theta \leftarrow \theta + \gamma \nabla l(\theta)$$

# Gradient of Log-Likelihood

---

Find the gradient of the log likelihood of the training corpus:

$$l(\theta) = \log \prod_i P(Y^{(i)} | X^{(i)})$$

# Interpretation of Gradient

---

Overall gradient is the difference between:

$$\sum_i \sum_t f_k(y_t^{(i)}, y_{t-1}^{(i)}, x_t^{(i)})$$

the empirical distribution of feature  $f_k$  in the training corpus

and:

$$\sum_i \sum_t \sum_{y, y'} f_k(y, y', x_t^{(i)}) P(y, y' | X^{(i)})$$

the expected distribution of  $f_k$  as predicted by the current model

# Interpretation of Gradient

---

When the corpus likelihood is maximized, the gradient is zero, so the difference is zero.

Intuitively, this means that finding parameter estimate by gradient descent is equivalent to telling our model to predict the features in such a way that they are found in the same distribution as in the gold standard.

# Regularization

---

To avoid overfitting, we can encourage the weights to be close to zero.

Add term to corpus log likelihood:

$$l^*(\theta) = \log \prod_i P(Y^{(i)} | X^{(i)}) - \sum_k \frac{\theta_k^2}{2\sigma^2}$$

$\sigma$  controls the amount of **regularization**

Results in extra term in gradient:

$$-\frac{\theta_k}{\sigma^2}$$

# Stochastic Gradient Descent (SGD)

---

In the standard version of the algorithm, the gradient is computed over the entire training corpus.

- Weight update only once per iteration through training corpus.

**Alternative:** calculate gradient over a small mini-batch of the training corpus and update weights

**SGD** is when mini-batch size is one.

- Many weight updates per iteration through training corpus
- Usually results in much faster convergence to final solution, without loss in performance



# Stochastic Gradient Descent

---

Goal: Minimize  $-l^*(\theta)$

Initialize  $\theta = \{\theta_1, \theta_2, \dots, \theta_k\}$  randomly

Do for a while:

- Randomize order of samples in training corpus

- For each mini-batch (of size one) in the training corpus:

  - Compute  $\nabla l^*(\theta)$  over this mini-batch

  - $\theta \leftarrow \theta + \gamma \nabla l^*(\theta)$