

This lecture discusses some variants of the basic Hidden Markov Model formulation in addition to a brief discussion of continuous random fields. These are another graphical model, about five years old, which are very similar to Hidden Markov Models, but with some key differences. This part of the lecture is a bit more shallow than the field deserves, but going much beyond this is outside the scope of this course. The reason for covering the topic in the first place is because it is a recent development in machine learning. Hence, we are among the first people in bioinformatics to learn about Continuous Random Fields!



Two learning scenarios

1. Estimation when the “right answer” is known

Examples:

GIVEN: a genomic region $x = x_1 \dots x_{1,000,000}$ where we have good (experimental) annotations of the CpG islands

GIVEN: the casino player allows us to observe him one evening, as he changes dice and produces 10,000 rolls

2. Estimation when the “right answer” is unknown

Examples:

GIVEN: the porcupine genome; we don't know how frequent are the CpG islands there, neither do we know their composition

GIVEN: 10,000 rolls of the casino player, but we don't see when he changes dice

QUESTION: Update the parameters θ of the model to maximize $P(x|\theta)$

CS262 Lecture 7, Win06, Batzoglou

Let us first review Hidden Markov Models. There are two main scenarios. The first is when the labels are given. That means we have a set of sequences, and a set of “true parses” which have generated these sequences. We do learning by looking at the underlying sequence of states, and count up the number of times each transition occurs, and each emission occurs. And based on these frequencies, we normalize them to be probabilities, and adjust the parameters of the model to reflect these frequencies.

The second case of learning is when there is no underlying sequence of states to rely on. In this case, we do something similar, but instead of calculating the actual counts of times each transition or emission is taken, we calculate the expected number of times each transition or emission is taken, given our current parameters. We do this by running the forward and backward algorithm, and then estimating those expectations. Then, we adjust the parameters of the model to reflect those expectations. Finally, we are guaranteed by the Expectation Maximization Theorem (which will not be covered), that the resulting probability of the data and the parse is higher ($p(x, \pi)$ is higher than with the previous parameters). We iterate this algorithm over and over, until convergence to a

new set of parameters. This new set of parameters gives us higher likelihood on the training examples, but is not necessarily optimal.



1. When the “true” parse is known

Given $x = x_1 \dots x_N$
for which the true $\pi = \pi_1 \dots \pi_N$ is known,

Simply count up # of times each transition & emission is taken!

Define:

A_{kl} = # times $k \rightarrow l$ transition occurs in π
 $E_k(b)$ = # times state k in π emits b in x

We can show that the maximum likelihood parameters θ (maximize $P(x|\theta)$) are:

$$a_{kl} = \frac{A_{kl}}{\sum_i A_{ki}}$$

$$e_k(b) = \frac{E_k(b)}{\sum_c E_k(c)}$$

CS262 Lecture 7, Win06, Batzoglou

Above are the formulas for updating new parameters, if we have labeled examples.



2. When the “true parse” is unknown

Baum-Welch Algorithm

Compute expected # of times each transition & is taken!

Initialization:

Pick the best-guess for model parameters
(or arbitrary)

Iteration:

1. Forward
2. Backward
3. Calculate $A_{kl}, E_k(b)$, given θ_{CURRENT}
4. Calculate new model parameters $\theta_{\text{NEW}} : a_{kl}, e_k(b)$
5. Calculate new log-likelihood $P(x | \theta_{\text{NEW}})$

GUARANTEED TO BE HIGHER BY EXPECTATION-MAXIMIZATION

Until $P(x | \theta)$ does not change much

CS262 Lecture 7, Win06, Batzoglou

Above are the formulas for updating new parameters, if we do not have labeled examples, using the Baum-Welch algorithm.

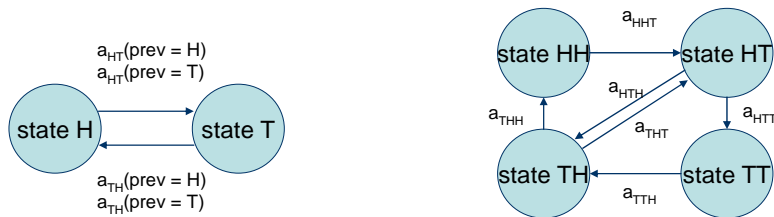
Variants of HMMs

We have discussed some variants of Hidden Markov Models.



Higher-order HMMs

- How do we model “memory” larger than one time point?
- $P(\pi_{i+1} = l \mid \pi_i = k)$ a_{kl}
- $P(\pi_{i+1} = l \mid \pi_i = k, \pi_{i-1} = j)$ a_{jkl}
- ...
- A second order HMM with K states is equivalent to a first order HMM with K^2 states

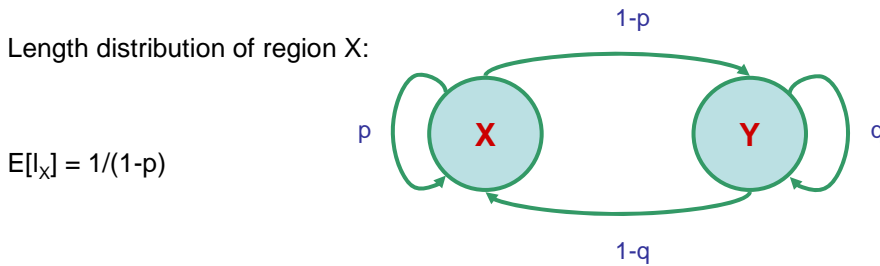


CS262 Lecture 7, Win06, Batzoglou

We discussed Higher-Order HMMs.



Modeling the Duration of States



- Geometric distribution, with mean $1/(1-p)$

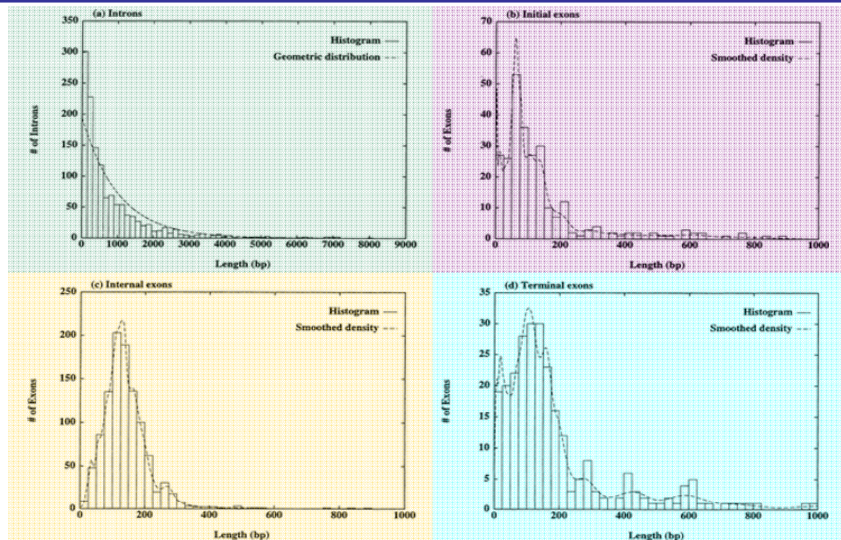
This is a significant disadvantage of HMMs

Several solutions exist for modeling different length distributions

CS262 Lecture 7, Win06, Batzoglou

Then, we talked about the fundamental deficiency with the HMM model. That is, the states have an underlying geometric distribution on their length. That means that the model in principle cannot capture more complicated, or different distributions on the duration spent in each state.

Example: exon lengths in genes

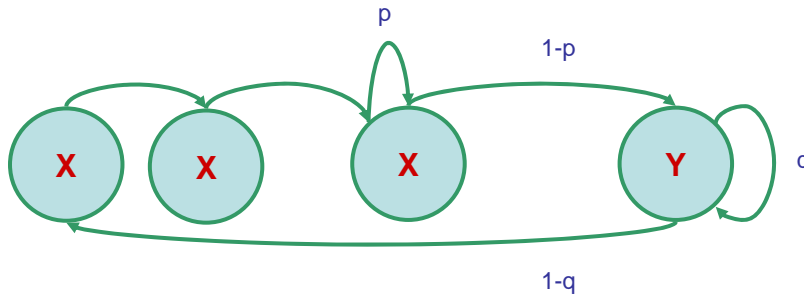


CS262 Lecture 7, Win06, Batzoglou

For example, when we do gene finding, we may want to have a different state for each kind of region within which we are. That is, we may want to have a state for introns, a state for exons, and in particular, a state for beginning exon, internal exon or for a gene that is a single exon without introns. Looking at the real data, we observe that introns tend to have a distribution that appears to be geometric to the naked eye. Exons, however, have a very different distribution than geometric, because there is a minimum length in general. There are very few exons that have a very short length, and there is a minimum length in which the typical exon can be found. So, what you find is some distribution that has some peak, then goes down geometrically, though perhaps with a longer tail. Now, if we model Exons with a regular state model, the idea is that we might be over penalizing (in terms of probability) Exons that are at the peak of the distribution, while favoring Exons that are too short, and perhaps penalizing the long tail (Exons that are too long). In summary, we are fitting an incorrect probability distribution.

How do we fix this problem?

Solution 1: Chain several states



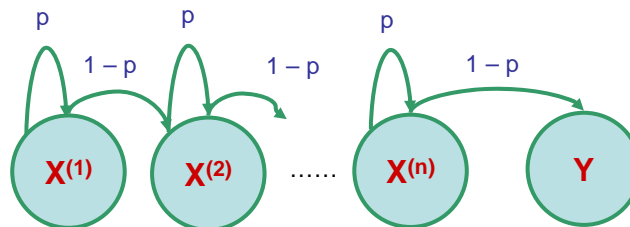
Disadvantage: Still very inflexible

$$I_X = C + \text{geometric with mean } 1/(1-p)$$

CS262 Lecture 7, Win06, Batzoglou

One idea is to chain many copies of a given state and require the model to pass through many copies before it goes to another state. That will ensure that we have a minimum length equal to that many copies. This particular distribution shown above is not very rich. It is basically a constant plus a geometric with the same mean as before.

Solution 2: Negative binomial distribution



Duration in **X**: m turns, where

- During first $m - 1$ turns, exactly $n - 1$ arrows to next state are followed
- During m^{th} turn, an arrow to next state is followed

Here is another more interesting variant. We can conveniently create a negative binomial distribution for the duration of a given state. In the above example, we are taking a given state and creating n copies. Each copy has a self-transition with probability p , and a transition to the next copy with probability $1-p$. The n^{th} copy has a probability p transition to itself and a probability $1-p$ transition to some state Y (or many other states). What is the probability distribution of lengths in which we spend the entire set of copies of X ? We have the calculation at the bottom of the figure.

We spend m turns in X , where m must be greater than n . So, during the first $m-1$ turns, we have to have exactly $m-1$ arrows to a next state. During the first $m-1$ turns, somewhere we had $n-1$ jumps from one state to the next. And in the n^{th} term, we got out of the state X .

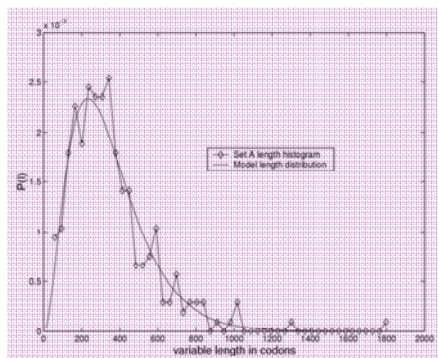
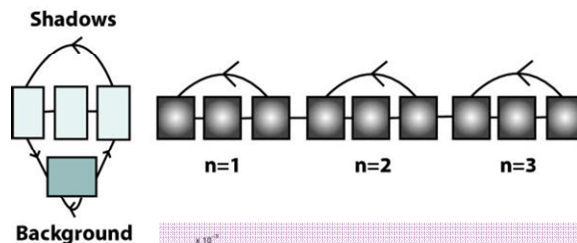
Every time we take an arrow to a next state, we multiply by the probability $1-p$; every time we take an arrow to the same state, we multiply by the probability p . Now, we calculate how many different combinations there are, in which we have $n-1$ arrows to the next state in $m-1$ turns, then one arrow to the last turn. So, the probability that we spend exactly m turns in all of the states X is:

$$P(l_x = m) = \binom{m-1}{n-1} (1-p)^{n-1+1} p^{(m-1)-(n-1)} = \binom{m-1}{n-1} (1-p)^n p^{m-n}$$

Example: genes in prokaryotes



- EasyGene: Prokaryotic gene-finder
Larsen TS, Krogh A
- Negative binomial with $n = 3$



Above is the plot of the function P . It looks very similar to an Exon distribution. The above example has three internal Exon states, the first two of which would link to the next state, and the last would link to Intron states. This yields a distribution which, by

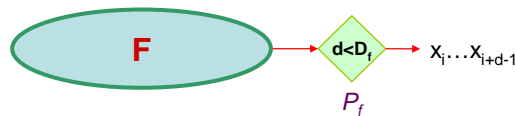
eye, looks like an Exon distribution. This method has been implemented in EasyGene Prokaryotic gene-finder, which is a gene finder for microorganisms. This is a nice trick for generating a negative binomial distribution (if that is the desired distribution). Now, we go on to doing more general distributions. This is explicit duration modeling, which also leads to what is also traditionally called “Semi Hidden-Markov Models.”

Solution 3: Duration modeling



Upon entering a state:

1. Choose duration d , according to probability distribution
2. Generate d letters according to emission probs
3. Take a transition to next state according to transition probs



Disadvantage: Increase in complexity of Viterbi:

Time: $O(D)$
Space: $O(1)$

Warning, Rabiner's tutorial claims $O(D^2)$ & $O(D)$ increases

where D = maximum duration of state

CS262 Lecture 7, Win06, Batzoglou

Upon entering a state, you first choose a duration with an explicit probability distribution. In order to make the algorithms more efficient, you also choose a maximum duration for this state. Let's refer to this as D_f , for state f . Then, once you choose this duration, d , you know you will spend exactly d steps in this state. You then emit d symbols, and move to a new state.

We generate exactly d letters, $x_i \dots x_{i+d-1}$. Then, we are obliged to take a transition to a different state than X . So, this is a significantly different model than the traditional HMMs that we discussed.

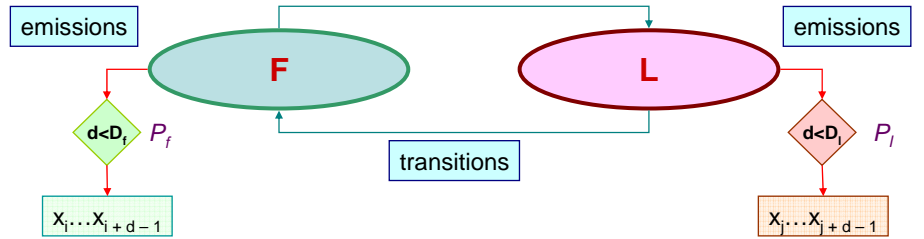
How do we do all of the HMM algorithms in this model?

Everything carries forward in a pretty straight-forward way, but the algorithms slow down. Viterbi, for instance, becomes D times slower, where D is the maximum duration of a given state. So, if you want to model Exons this way, and you want your Exons to be length at most 2000, you end up with an algorithm that is 2000-times slower. However, in practice, there are many ways of speeding things up, and not needing to consider the entire Dynamic Programming matrix.

Note: Rabiner's tutorial, the classic reference for HMMs, because some tricks were overlooked, claims that there is an $O(D^2)$ time-penalty (and an additional $O(D)$ space penalty), which is incorrect.

Implementing Viterbi with duration modeling

Viterbi with duration modeling



We have our example of a dishonest casino player, which has a fair and a loaded die. Our casino player starts in one of the two states. Now, even before the player rolls once, he chooses the length of time he will spend in a given state from some distribution. After that number, he necessarily will switch dies. How can we find an optimal parse of a given set of rolls?

Recall the original iteration function for Viterbi:

$$V_l(i) = \max_k V_k(i-1) a_{kl} \cdot e_l(x_i)$$

The reason this was possible was because every decision to either switch or not switch from F to L was independent of all previous decisions.

Now, we are in position i . We assume that position was the last position emitted from a given state. So, for this example, position i was the last fair or last loaded roll. We do it as follows:

$$V_l(i) = \max_k \max_{d=1 \dots D_l} V_k(i-d) \cdot P_l(d) \cdot a_{kl} \cdot \prod_{j=i-d+1 \dots i} e_l(x_j)$$

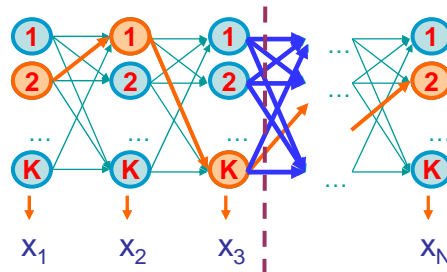
To explain the above, realize that we were just in state k , and we are about to enter state l .

For every possible value of d , it seems that we do d amounts of work, since we calculate the product $\prod_{j=i-d+1..i} e_l(x_j)$. However, in linear time, we can calculate the cumulative product, or the cumulative sum of logs, for every state. We can keep the cumulative product, then every time we desire to do the product calculation, we divide the last position by one-before the first position (or subtract the corresponding logarithms). As such, instead of doing the product step in $O(d)$ time, it is done in constant time. The precomputation itself is done in linear time for every state, which is asymptotically less time than the running time of the rest of the entire algorithm.

Therefore, going back to our example, we take an $O(2000)$ hit, which is not terrible. But, we can now calculate the probability distributions for lengths up to 2000. Granted, there are various things that can be done to speed up this process, if time is an issue. This is especially important if one wants to train data using this model, in which case one would need to run Forward and Backward repeatedly in a similar modified fashion to the modification of Viterbi, until convergence.

Conditional Random Fields

Let's look at an HMM again



Why are HMMs convenient to use?

- Because we can do dynamic programming with them!
 - "Best" state sequence for $1..i$ interacts with "best" sequence for $i+1..N$ using K^2 arrows

$$V_i(i+1) = e_i(i+1) \max_k V_k(i) a_{ki}$$

$$= \max_k (V_k(i) + [e(i, i+1) + a(k, i)]) \quad (\text{where } e(\cdot, \cdot) \text{ and } a(\cdot, \cdot) \text{ are logs})$$

- Total likelihood of all state sequences for $1..i+1$ can be calculated from total likelihood for $1..i$ by only summing up K^2 arrows

Why are hidden markov models convenient to use?

One thing that is extremely convenient is that one can easily implement and run, with very efficient computation times, their algorithms for parsing and for learning. So, basically, it is because we can run DP-like algorithms for finding the most likely parse of a given sequence, and also for estimating the sum of probabilities of parses, and finding new, good parameters for the model.

Let us look at Viterbi again. The reason we can do dynamic programming is because at any given time point, the best state sequence for the first i letters interacts with the best state sequence for the first $i+1$ letters only through k^2 arrows, where k is our number of states. Our decision about what we do in the first i states does not affect what we do in the $i+1$ state, except that we are ending up in the first i steps in a given state. So, all we have to do to calculate the $i+1$ position is to look at all the possibilities for position i .

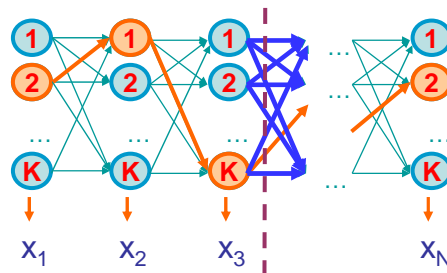
We rewrite the Viterbi formula in a way that is more useful for the new model.

$$V_l(i+1) = e_l(i+1) \max_k V_k(i) a_{kl} = \max_k (V_k(i) + [e_l(i+1) + a(k,l)])$$

Note that the last part of the equation is done in log-form.

We can perform a similar operation on Forward and Backward.

Let's look at an HMM again



- Some shortcomings of HMMs
 - Can't model state duration
 - Solution: explicit duration models (Semi-Markov HMMs)
 - Unfortunately, state π_i cannot "look" at any letter other than x_i !
 - Strong independence assumption: $P(\pi_i | x_1 \dots x_{i-1}, \pi_1 \dots \pi_{i-1}) = P(\pi_i | \pi_{i-1})$

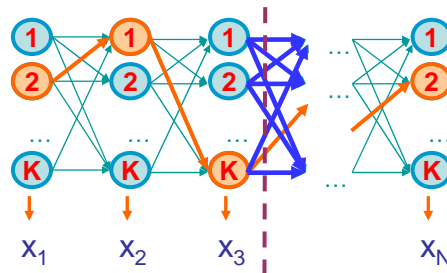
CS262 Lecture 7, Win06, Batzoglu

Here are some shortcomings of HMMs. One such shortcoming is the Geometric Distribution on length. We addressed this with Semi-Markov HMMs, or other tricks such as the negative-binomial distribution.

Another shortcoming is that HMMs are very limited in what they can model. A given state cannot affect anything in the sequence except the current letter x_i . This is unreasonable in our standard Casino Player model. However, in the real world of biology, things are not that simple. A given Exon may be more likely to be functional, depending on context. Namely, if we have CpG islands in neighboring regions, then we may have higher likelihood.

The reason this cannot be done is because of the strong dependence assumption that the probability for the current state given everything that has happened before is equal to the probability for the current state given the previous state.

Let's look at an HMM again



- Another way to put this, features used in objective function $P(x, \pi)$:
 - $a_{kl}, e_k(b)$, where $b \in \Sigma$
 - At position i : all K^2 a_{kl} features, and all K $e_k(x_i)$ features play a role
 - OK forget probabilistic interpretation for a moment
 - “Given that prev. state is k , current state is l , how much is current score?”
 - $V_l(i) = V_k(i-1) + (a(k, l) + e(l, i)) = V_k(i-1) + g(k, l, x_i)$
 - Let's generalize $g!!!$ $V_k(i-1) + g(k, l, \mathbf{x}, i)$

CS262 Lecture 7, Win06, Batzoglou

Another way to look at this is that the features used in the objective function $P(x, \pi)$ to be maximized are transition probabilities between pairs of states and emission probabilities of given letters. At every position i , when we do Viterbi, we take into account all k^2 features concerning transition probabilities, and all k emission probability features $e_l(x_i)$.

From this point forward, we will drop the probabilistic interpretation of Hidden Markov Models. We will just think of them as predictors of the underlying sequences of states. We are trying to maximize the score -- $P(x, \pi)$ (think of this as a score, not a probability). The score is a function of the features, the emission and transition probabilities.

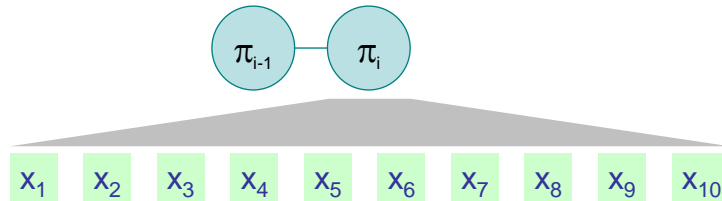
So, given that the previous state is k and the current state is l , at position i , what is the current score? We define the function g as follows.

$$V_l(i) = V_k(i-1) + (a(k, l) + e(l, i)) = V_k(i-1) + g(k, l, x_i)$$

We generalize and extend the definition of g to include the value i as well as the entire sequence x .

Thus, we allow the objective function to look at the entire sequence at once.

“Features” that depend on many pos. in x



- What do we put in $g(k, l, x, i)$?
 - The “higher” $g(k, l, x, i)$, the more we like going from k to l at position i
- Richer models using this additional power
 - Examples
 - Casino player looks at previous 100 pos'ns; if > 50 6s, he likes to go to Fair
 $g(\text{Loaded}, \text{Fair}, x, i) += \mathbf{1}[x_{i-100}, \dots, x_{i-1} \text{ has } > 50 \text{ 6s}] \times w_{\text{DONT_GET_CAUGHT}}$
 - Genes are close to CpG islands; for any state k ,
 $g(k, \text{exon}, x, i) += \mathbf{1}[x_{i-1000}, \dots, x_{i+1000} \text{ has } > 1/16 \text{ CpG}] \times w_{\text{CG_RICH_REGION}}$

CS262 Lecture 7, Win06, Batzoglou

Intuitively, at any position, the state in which we are will depend on our previous state and the entire sequence x . Thus, in the example of Introns and Exons, we may now look not only at the previous state, which was an intron, in addition to the current letter, but we can look at the entire area around the current position. Perhaps there are CpG islands. Perhaps there is some other Gene nearby. Perhaps there is a transcription factor binding site.

Intuitively, we consider the function g to be a “score.” The higher the score, the more we want to go to state l from state k in position i of sequence x . The lower the score, the more we want to avoid this transition.

We can model very rich features with such a score. For example, perhaps our casino player is more sophisticated than previously believed in prior lectures. Perhaps the Casino player is afraid to be caught. Perhaps in the past 100 rolls, if the player has more than 50 6's, then this will influence the player's decision to select a die.

Consider another example, this time having to do with Biology. Perhaps we scan an entire genomic region, and find clusters of CpG islands. Then, in the vicinity of CpG islands, we boost the score of Exons, since genes tend to occur more commonly within CpG islands.

How do we do this computationally? Using Conditional Random Fields!

“Features” that depend on many pos. in x



x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10}

Conditional Random Fields—Features

1. Define a set of features that you think are important
 - All features should be functions of current state, previous state, x , and position i
 - Example:
 - Old features: transition $k \rightarrow l$, emission b from state k
 - Plus new features: prev 100 letters have 50 6s
 - Number the features $1 \dots n$: $f_1(k, l, x, i), \dots, f_n(k, l, x, i)$
 - features are indicator true/false variables
 - Find appropriate weights w_1, \dots, w_n for when each feature is true
 - weights are the parameters of the model

2. Let's assume for now each feature has a weight w_j

- Then, $g(k, l, x, i) = \sum_{j=1 \dots n} \mathbf{1}[\text{feature } j \text{ is true at pos } i] \times w_j$
 $= \sum_{j=1 \dots n} f_j(k, l, x, i) \times w_j$

CS262 Lecture 7, Win06, Balzoglou

Conditional Random Fields allow you to define a set of features that you may consider important. Those features are not as constrained as HMMs, which can only be the likelihoods that you change states and emit letters from particular states. The features are basically anything imaginable, influencing a state from context, with one important restriction: all the features should be functions of the current state, the previous state, any position in the sequence x , and the current position's index.

Conditional Random Fields allow us to model all of the old features, by scaling down the amount of power we are using from the CRFs.

Alternatively, we can use the new power to have the CRF look at the previous 100 rolls (in the casino player example) and make a decision based on whether there have been 50 6's or more.

It's convenient to think of every kind of feature as an indicator variable. So, we will number our features, and call them feature 1 up to feature n . We will consider them as true/false features. They are 1 if a given thing is true in the current position, and 0 otherwise. So, if we are given a sequence and an underlying sequence of states, transition k_l is true in position i if indeed we are in state l and the previous state is k .

Each feature has a weight associated with it. So, if each feature $f_1 \dots f_n$, has a weight $w_1 \dots w_n$ which can be an arbitrary value, positive or negative. Those weights are the parameters of the model. So, for each feature, we have a parameter in the model. Now, you will notice, HMMs can be very easily described this way. Every transition and every emission is a feature. The weight w are simply the logarithms of those probabilities. However, HMMs are a limited use of this more powerful model.

The objective function that we look to maximize is the following:

$$g(k, l, x, i) = \sum_{j=1..n} 1[\text{feature } j \text{ is true at position } i] \cdot w_j = \sum_{j=1..n} f_j(k, l, x, i) w_j$$

So, at every position, we have a score, which depends on the previous state, the current state, x and the position. Intuitively, this means that every feature is really true or false, and every weight is just a positive or negative value.

“Features” that depend on many pos. in x



x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10}

Define

$V_k(i)$: Optimal score of “parsing” $x_1 \dots x_i$ and ending in state k

Then, assuming $V_k(i)$ is optimal for every k at position i , it follows that

$$V_k(i+1) = \max_k [V_k(i) + g(k, l, x, i+1)]$$

Why?

Even though at pos'n $i+1$ we “look” at arbitrary positions in x , we are not affected by the “decisions” of parsing positions $1, \dots, i-1$

Therefore, Viterbi algorithm again finds optimal (highest scoring) parse for $x_1 \dots x_N$

CS262 Lecture 7, Win06, Batzoglou

How do we find an optimal parse?

We want to find an underlying sequence of states that maximizes the sum of the scores in each position. So, assuming that just like before, V_k is optimal for every k at position i , we still have:

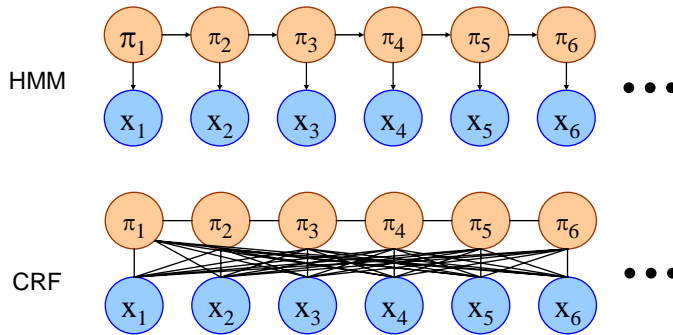
$$V_k(i+1) = \max_k [V_k(i) + g(k, l, x, i+1)]$$

Why does this work? What is the difference between this and the old Viterbi algorithm?

Otherwise, we would have a different parse. Basically, even though at position $i+1$ we look at an arbitrary number of positions in the sequence, and in particular the positions $1 \dots i$, we don't look at the underlying states of those positions. We only look at the sequence x , which is always constant. In fact, what we could do is for every position in the sequence and for every pair of states, we could precompute the table g . This will take time $O(k^2 n)$, where n is the length of the sequence. Then, we find the optimal Viterbi

parse, given this table. Even though the state for every position depends on the entire sequence, it only depends on the previous state. It does not depend on any other state. Hence, we can apply DP, specifically Viterbi.

“Features” that depend on many pos. in x



- Score of a parse depends on all of x at each position
- Can still do Viterbi because state π_i only “looks” at prev. state π_{i-1} and the constant sequence x

CS262 Lecture 7, Win06, Batzoglou

Consider the illustration above. In HMMs, every state emits a letter, then goes to the next state, and repeats this process. In CRFs, every state only interacts with the next state, but interacts with the entire sequence x . Of course, when we choose a set of features, we may choose to not have the CRF be so complicated. We may choose the entire process to be a lot more local, but this is not required.

How many parameters are there, in general?



- Arbitrarily many parameters!
 - For example, let $f_j(k, l, x, i)$ depend on $x_{i-5}, x_{i-4}, \dots, x_{i+5}$
 - Then, we would have up to $K \times |\Sigma|^{11}$ parameters!
 - **Advantage:** powerful, expressive model
 - **Example:** “if there are more than 50 6’s in the last 100 rolls, but in the surrounding 18 rolls there are at most 3 6’s, this is evidence we are in Fair state”
 - **Interpretation:** casino player is afraid to be caught, so switches to Fair when he sees too many 6’s
 - **Example:** “if there are any CG-rich regions in the vicinity (window of 2000 pos) then favor predicting lots of genes in this region”
 - **Question:** how do we train these parameters?

CS262 Lecture 7, Win06, Batzoglou

How many parameters are there in such a model, in general?

In CRFs, even if we have a very limited number of states, we may still have a huge number of parameters. For example, let a set of features depend on positions 5-before up to 5-after our current position. Then, we would have up to $K \cdot |\Sigma|^{11}$ parameters.

In general, CRFs allow us to train models with huge numbers of parameters, and are extremely powerful.

Conditional Training



- Hidden Markov Model training:
 - Given training sequence x , “true” parse π
 - Maximize $P(x, \pi)$
- Disadvantage:

- $P(x, \pi) = P(\pi | x) P(x)$

Quantity we care about
so as to get a good parse

Quantity we don't care so much
about because x is always given

CS262 Lecture 7, Win06, Batzoglou

Training the features is a fairly involved topic, so it is not covered in its entirety in this lecture.

For HMMs, given a training sequence x and a “true” parse π , we maximize $P(x, \pi)$. In fact, if we have sequences and underlying sequences of states, we find the optimum by doing a simple calculation of frequencies and adjust the weights accordingly.

What does this process actually do when we maximize $P(x, \pi)$? This maximizes the conditional probability $P(\pi | x) \cdot P(x)$. The quantity that we really care to maximize is $P(\pi | x)$. We do not care about the value $P(x)$, since we are given the mouse genome, the casino player's rolls, etc. However, we end up maximizing the value regardless.

Conditional Training



$$P(x, \pi) = P(\pi | x) P(x)$$
$$P(\pi | x) = P(x, \pi) / P(x)$$

Recall

$$F(j, x, \pi) = \# \text{ times feature } f_j \text{ occurs in } (x, \pi)$$
$$= \sum_{i=1 \dots N} f_j(k, l, x, i); \quad \text{count } f_j \text{ in } x, \pi$$

In HMMs, let's denote by w_j the weight of j^{th} feature: $w_j = \log(a_{kl})$ or $\log(e_k(b))$

Then,

$$\text{HMM:} \quad P(x, \pi) = \exp\left[\sum_{j=1 \dots n} w_j \times F(j, x, \pi)\right]$$

$$\text{CRF:} \quad \text{Score}(x, \pi) = \exp\left[\sum_{j=1 \dots n} w_j \times F(j, x, \pi)\right]$$

CS262 Lecture 7, Win06, Batzoglou

Instead of doing the previous training algorithm, Conditional Random Fields use “Conditional Training.” What this does is to maximize the conditional probability $P(\pi | x)$ directly. The idea is that we want to find parameters that are optimal for parsing. Namely, given new sequences, we will get the highest likelihood parses, given the sequences.

How is this accomplished in CRFs?

Recall the notation introduced in Lecture 4, where $F(j, x, \pi)$ represented the number of times the j^{th} feature of f occurs in sequence x and parse π , where a feature is a transition or emission probability. This can be taken as the sum, over all positions in x , of the indicator variable of whether this feature occurs exactly in this position.

Let us denote by w_j the logarithm of the transition or emission probability associated with the feature j . Then $w_j = \log(a_{kl})$ or $\log(e_k(b))$. Then, in HMMs:

$$P(x, \pi) = e^{\sum_{i=1 \dots n} w_i \cdot F(i, x, \pi)}$$

For CRFs, we have not talked about probabilities, but about scores:

$$\text{Score}(x, \pi) = e^{\sum_{i=1 \dots n} w_i \cdot F(i, x, \pi)}$$

Conditional Training



In HMMs,

$$P(\pi | x) = P(x, \pi) / P(x)$$

$$P(x, \pi) = \exp\left[\sum_{j=1 \dots n} w_j \times F(j, x, \pi)\right]$$

$$P(x) = \sum_{\pi} \exp\left[\sum_{j=1 \dots n} w_j \times F(j, x, \pi)\right] =: Z$$

Then, in CRF we can do the same to normalize $\text{Score}(x, \pi)$ into a prob.

$$P_{\text{CRF}}(\pi | x) = \exp\left[\sum_{j=1 \dots n} w_j \times F(j, x, \pi)\right] / Z$$

QUESTION: Why is this a probability???

CS262 Lecture 7, Win06, Batzoglu

We define $Z := P(x)$. We can calculate this using the Forward/Backward algorithms.

In CRFs, we can do the same thing as in HMMs. We can normalize the score of x, π and turn it into a probability. So, the probability that the CRF gives for a parse, given a

sequence x , will be defined as:
$$\frac{e^{\sum_{j=1 \dots n} w_j \cdot F(j, x, \pi)}}{Z}$$
.

Why is this a probability distribution?

Because in CRFs, this is the quotient of a positive number and its normalization.

Conditional Training



1. We need to be given a set of sequences x and “true” parses π
2. Calculate Z by a sum-of-paths algorithm similar to HMM
 - We can then easily calculate $P(\pi | x)$
3. Calculate partial derivative of $P(\pi | x)$ w.r.t. each parameter w_j
(*not covered—akin to forward/backward*)
 - Update each parameter with gradient descent!
4. Continue until convergence to **optimal** set of weights

$$P(\pi | x) = \exp\left[\sum_{j=1\dots n} w_j \times F(j, x, \pi)\right] / Z \text{ is convex!!!}$$

CS262 Lecture 7, Win06, Balzoglou

We train CRFs as follows. We need to be given a set of sequences and true parses for them. In the casino player example, this is a set of rolls, in addition to which rolls were produced by the fair or loaded dies. Then, we calculate the value of Z by a sum-of-paths algorithm, which is similar to HMMs. This is simply a dynamic programming algorithm, that will calculate all true positions of the sequence x . Then, the next step (which is not covered in this lecture), is to calculate the partial derivative of $P(\pi | x)$ with respect to each parameter w_j . We want to maximize this quantity, so we apply partial derivatives to essentially perform gradient descent. We continue updating until convergence. This relies on the objective function being convex. Otherwise, we would not necessarily come across a global optimum. The parameters that we find are considered best for predictions, but not best for modeling the sequence.

Conditional Random Fields—Summary



1. Ability to incorporate complicated non-local feature sets
 - Do away with some independence assumptions of HMMs
 - Parsing is still equally efficient
2. Conditional training
 - Train parameters that are best for *parsing*, not *modeling*
 - *Need* labeled examples—sequences x and “true” parses π
(*Can train on unlabeled sequences, however it is unreasonable to train too many parameters this way*)
 - Training is significantly slower—many iterations of forward/backward

CS262 Lecture 7, Win06, Batzoglou

In summary, there are two main advantages of CRFs over HMMs. Clearly, every HMM is a CRF, so there are no real advantages of HMMs over CRFs. First, you are allowed to incorporate much more complicated features into CRFs. Hence, much more involved considerations are allowed when taking the parse of a sequence. Secondly, you are training the parameters of a model, in order not to reflect the underlying sequences that are generated, but you are using the model to predict the underlying sequence of states.

In general, Conditional Random Fields is a new method that is considered to be more accurate than Hidden Markov Models, if they are built and trained properly.