BWT Continuation and Hidden Markov Models

CS262 Lecture 5 Notes

January 19, 2016

Scribe: Pavitra Rengarajan

1 Applications of Burrows-Wheeler Transform

Recall that the Burrows-Wheeler Transform is a reversible means of permuting characters of a string S to another string BWT(S), and from BWT(S) to the original string S. In lecture 4, we covered the use of BWT in finding exact alignments through use of the suffix array. The algorithm, which was published by Michael Burrows and David Wheeler in 1994, serves as a text transform that often improves compressibility and increases search efficiency. Modern fast read aligners such as BWA, Bowtie, and SOAP are all based on the Burrows-Wheeler transform, and strive to answer the following seemingly difficult problem: *Given strings S and w, where $|S| >> w$, find all occurrences of w in S in $O(|w|)$ time.*

1.1 BWA Inexact Match

Since very few positions will have large sequence diversions from the query to the reference, we want to be able to allow a few sequencing errors and base differences. The Burrows-Wheeler Alignment tool (BWA) efficiently aligns short sequencing reads against a large reference sequence such as the human genome, while allowing mismatches and gaps. In practical applications of BWT such such as BWA and Bowtie (described below), inexact matches are needed, and BWT is used as a subroutine.

Suppose we would like to allow up to n mismatches or gaps. Given a read W, we keep track of multiple partial alignments, each one of which is associated with a different range pointing to the suffix array. The partial alignments are 4-tuples of (i, z, L, U), where i is the current position, z is the remaining non-matches allowed, L is the current low, and U is the current high. The backward search algorithm is run as follows:

```
I ← 0
I ← I ∪ InexRecur(W, i-1, z-1, k, l)
for each b ∈ {A, C, G, T} do
  k ← C(b) + O(b, k-1) + 1
  l ← C(b) + O(b,l)
  if k ≤ l then
    I ← I ∪ InexRecur(W, i, z-1, k, l)
    if b = W[i] then
      I ← I ∪ InexRecur(W, i-1, z, k, l)
    else
      I ← I ∪ InexRecur(W, i-1, z-1, k, l)
```

This procedure returns the intervals of substrings that match W with no more than z differences (mismatches or gaps); `InexRecur(W,i,z,k,l)` recursively calculates the intervals of substrings that match $W[0, i]$ with no more than z differences on the condition that suffix W_{i+1} matches interval $[k, l]$. Note that if there is a gap in the reference, the range has not changed. If there is a gap in the read, we do not decrease i because we are still going to process that read's letter next, and in the reference we search through all of the possible 4 letters that could align to this gap.

A study by Heng Li and Richard Durbin found that BWA is on the order of 10 to 20 times faster than MAQ, which is a hash table-based read alignment method. In theory, BWA works with arbitrarily long reads, but its performance in practice is degraded on long reads, especially when the sequencing error rate is high. Since BWA requires the full read to be aligned, BWA does not work as well on long reads, which are more likely to be disrupted by structural variations in the reference genome. Alternative read alignments are used in practice for long reads; for example, Li and Durbin suggest dividing the read into multiple short fragments, aligning the fragments separately with the aforementioned algorithm, then joining the partial alignments to get the full alignment of the read.

1.2 BWA Heuristics

Multiple heuristics could be used to capitalize on the sensitivity-speed tradeoff and increase efficiency. Examples include the following:

- Maintain a lower bound array D, where $D(i)$ is a lower bound on the number of differences of exactly matching $R[0, i]$ with the reference (this can be computed in $O(|R|)$ time)
- Process best partial alignments first using a structure like a min-priority heap to store alignment entries rather than recursion
- Prune out alignments that are considered suboptimal by stopping at a threshold
- Limit the number of differences in the seed sequence, or the first k base pairs
- Disallow indels at the ends of the read

1.3 Bowtie

Another application of the BWT is with an algorithm called Bowtie. In their paper, Langmead et al. describe Bowtie as an ultrafast, memory-efficient alignment program for aligning short DNA sequence reads to large genomes. Bowtie employs a quality-aware backtracking algorithm to extend BWT so as to permit certain mismatches. This extremely efficient algorithm works especially well for use cases such as mammalian re-sequencing.

2 Hidden Markov Models (HMMs)

On a fundamental level, the motivation behind HMMs comes from the fact that given a sequence, we would like to understand if that sequence has some sort of underlying structure. Each label on

different parts of the sequence (e.g. "this is a gene" or "this is not a gene") has statistical properties. We want to be able to model the statistical properties of sequences that are labeled and look at observable sequences to understand the hidden labels. HMMs have a wide array of applications, ranging from alignment of bio-sequences, protein folding, and gene prediction to part-of-speech tagging, speech synthesis, and machine translation.

2.1 Dishonest Casino Model

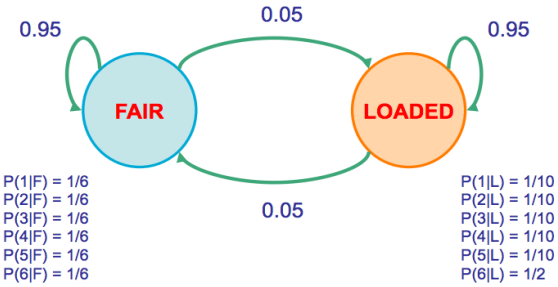
As an introduction to HMMs, we consider the dishonest casino model. A casino has two dice:

- *Fair die:* $P(1) = P(2) = P(3) = P(4) = P(5) = P(6) = 1/6$
- *Loaded die:* $P(1) = P(2) = P(3) = P(4) = P(5) = 1/10$, and $P(6) = 1/2$

Then, the game operates as follows:

- You bet \$1.
- You roll with a fair die.
- Casino player rolls with either fair die or loaded die.
- Highest number wins \$2.

In this scenario, fair and loaded are the hidden states of system. The sequence of rolls serve as the observable values. The HMM in this scenario looks as follows:



2.2 HMM Overview

HMMs strive to answer 3 core questions:

- **Evaluation:** Given a sequence of rolls by the casino player, how likely is this sequence, given our model of how the casino works?
- **Decoding:** Given a sequence of rolls by the casino player, what portion of the sequence was generated with the fair die, and what portion with the loaded die?

- **Learning:** Given a sequence of rolls by the casino player, how "loaded" is the loaded die? How "fair" is the fair die? How often does the casino player change from fair to loaded and back?

An HMM is **memory-less** because it is not necessary to consider previous rounds; just the current state determines the next state. Stated more formally, at each time step t, the only thing that affects future states is the current state π_t.

2.3 HMM Definition

A hidden Markov model (HMM) is defined by the following quantities:

- **Alphabet:** $\sum = \{b_1, b_2, ..., b_m\}$
- **Set of states:** $Q = \{1, ..., K\}$
- **Transition probabilities:** a_{ij} = transition probability from state i to state j, and $a_{i1} + ... + a_{iK} = 1$ for all states $i = 1...K$
- **Start probabilities:** a_{0i}, where $a_{01} + ... + a_{0k} = 1$
- **Emission probabilities:** $e_i(b) = P(x_i = b|\pi_i = k)$, where $e_i(b_1) + ... + e_i(b_K) = 1$ for all states $i = 1...K$

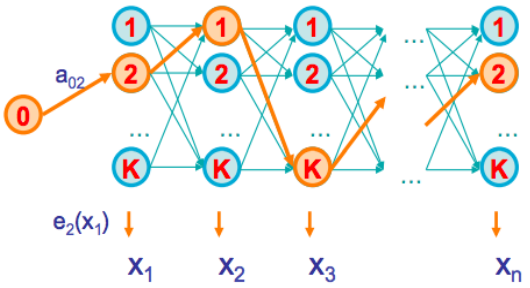
As aforementioned, an HMM is **memory-less**. The only thing that affects future states is the current state π_t for each time step t. In other words,

$P(\pi_{t+1} = k|\text{"whatever happened so far"}) =$
$P(\pi_{t+1} = k|\pi_1, \pi_2, ..., \pi_t, x_1, x_2, ..., x_t) =$
$P(\pi_{t+1} = k|\pi_t)$

2.4 Parses

Given a sequence $x = x_1, ..., x_N$, a **parse** of x is a sequence of states $\pi = \pi_1, ..., \pi_N$. Given an HMM, we can easily generate a sequence of length n by doing the following:

- Start at state π_1 according to probability $a_{0\pi 1}$
- Emit letter x_1 according to probability $e_{\pi 1}(x_1)$
- Go to state π_2 according to probability $a_{\pi 1 \pi 2}$
- Continue until emitting x_n



The **likelihood of a parse** can be calculated by taking the product of the transmission and emission probabiliities:

$P(x = x_1, ..., x_n, \pi = \pi_1, ..., \pi_n) = P(x_N|\pi_n)P(\pi_N|\pi_{N-1}...P(x_2|\pi_2)P(x_1|\pi_1)P(\pi_1))$

Let us consider the concrete example of our casino example HMM. We have a finite, often small number of parameters. In this model, we have 4 transition parameters, of which 2 are independent. We also have 12 emission probabilities (1-6 from each of 2 states), and 2 start probabilities. Thus, we have 18 parameters; let us label them from θ_1 to θ_{18}. If you play 1 million rolls, the sequence has 2 million arrows. We can approach this another way by creating an indicator called the feature counts F representing the number of times a parameter occurs. Then, we have the following:

$F(j, x, \pi)$ = number of times parameter θ_j occurs in (x, π)
$P(x, \pi) = \prod_{j=1...n} \theta_j^{F(j,x^\pi)} = \exp[\sum_{j=1...n} \log(\theta_j) * F(j, x, \pi)]$

2.5 Decoding

We now take a closer look at the decoding question of HMMs. Given a sequence, our objective is to find the most likely series of states. In otherwords, given $x = x_1, x_2, ..., x_N$, we want to find $\pi = \pi_1, ..., \pi_N$ that maximizes $P[x, \pi]$. We can denote this as $\pi^* = \text{argmax}_\pi P[x, \pi]$. A naive approach would involve an exhaustive search, but in order to consider k^n possibilities for some constant k, this could take an extremely long time. Thus, we explore a dynamic programming approach called the Viterbi algorithm.

2.5.1 Viterbi Algorithm

We operate with the following inductive assumption:

For all states k and for a fixed position i,
$V_k(i) = \text{max}_{\pi 1...\pi i-1} P[x_1...x_{i-1}, \pi_1, ..., \pi_{i-1}, x_i, \pi_i = k]$

The inductive step then follows:

$V_l(i+1) = \text{max}_{\pi 1...\pi i} P[x_1...x_{i-1}, \pi_1, ..., \pi_{i-1}, x_i, \pi_i = l]$
$= \text{max}_{\pi 1...\pi i} P(x_{i+1}, \pi_{i+1} = l|x_1...x_i, \pi_1, ..., \pi_i) P[x_1...x_i, \pi_1, ..., \pi_i]$
$= \text{max}_{\pi 1...\pi i} P(x_{i+1}, \pi_{i+1} = l|\pi_i) P[x_1...x_i, \pi_1, ..., \pi_i, x_i, \pi_i]$
$= \text{max}_k [P(x_{i+1}|\pi_{i+1} = l) P(\pi_{i+1} = l|\pi_i = k) V_k(i)] = e_l(x_{i+1}) \ \text{max}_k a_{kl} V_k(i)$

We now present a summary on the Viterbi algorithm:

- **Initialization:** $V_0(0) = 1$ and $V_k(0) = 0$ for all $k > 0$
- **Iteration:** $V_j(i) = e_j(x_i) \times \ \text{max}_k a_{kj} V_k(i-1)$ and $Ptr_j(i) = \text{argmax}_k a_{kj} V_k(i-1)$
- **Termination:** $P(x, \pi^*) = \text{max}_k V_k(N)$
- **Traceback:** $\pi_N^* = \text{argmax}_k V_k(N)$ and $]pi_{i-1}^* = Ptr_{\pi i}(i)$

- **Time complexity:** $O(K^2N)$
- **Space complexity:** $O(KN)$

Note a practical detail that in order to combat the problem of underflow (extremely small numbers), you can take the logs of all values and sum them.

2.6 Evaluation

Evaluation consists of finding the likelihood that a sequence is generated by the model. As such, we are attempting to find:

- $P(x)$, the probability of x given the model
- $P(x_i...x_j)$, the probability of a substring of x given the model
- $P(\pi_i = k|x)$, the posterior probability that the ith state is k given x

2.6.1 The Forward Algorithm

We are interested in calculating $P(x)$, the probability of x given the HMM. In order to avoid summing over an exponential number of paths π, we define the **forward probability** to be $f_k(i) = P(x_1...x_i, \pi_i = k)$, or the probability that we generate i first characters of x and end up in state k.

Derivation.

$f_k(i) = P(x_1...x_i, \pi_i = k)$
$= \sum_{\pi 1...\pi i-1} P(x_1...x_{i-1}, \pi_1, ..., \pi_{i-1}, \pi_i = k)e_k(x_i)$
$= \sum_l \sum_{\pi 1...\pi i-2} P(x_1...x_{i-1}, \pi_1, ..., \pi_{i-2}, \pi_{i-1} = l)a_{lk}e_k(x_i)$
$= \sum_l P(x_1...x_{i-1}, \pi_{i-1} = l)a_{lk}e_k(x_i)$
$= e_k(x_i) \sum_l f_l(i-1)a_{lk}$

Thus, we see that we can compute $f_k(i)$ for all k, i using dynamic programming. A summary of the forward algorithm follows:

- **Initialization:** $f_0(0) = 1$ and $f_k(0) = 0$ for all $k > 0$
- **Iteration:** $f_k(i) = e_k(x_i) \sum_l f_l(i-1)a_{lk}$
- **Termination:** $P(x) = \sum_k f_k(N)$

2.6.2 The Backward Algorithm

The motivation for the backward algorithm is that we want to compute $P(\pi_i = k|x)$, the probability distribution on the ith position, given x. We start by computing:

$P(\pi_i = k, x) = P(x_1...x_i, \pi_i = k, x_{i+1}...x_N) = P(x_1...x_i, \pi = k)P(x_{i+1}...xN|x_1...x_i, \pi_i = k)$
$= P(x_1...x_i, \pi = k)P(x_{i+1}...x_N|\pi = k)$.

The first term $P(x_1...x_i, \pi = k)$ is the forward probability $f_k(i)$, and the second term $P(x_{i+1}...x_N|\pi = k)$ is the backwards probability $b_k(i)$. We now derive a recursive expression for $b_k(i)$.

Derivation.

$b_k(i) = P(x_{i+1}...x_N|\pi_i = k)$
$= \sum_{\pi i+1...\pi N} P(x_{i+1}, x_{i+2}, ..., x_N, \pi_{i+1}, ..., \pi_N|\pi_i = k)$
$= \sum_l \sum_{\pi i+1...\pi N} P(x_{i+1}, x_{i+2}, ..., x_N, \pi_{i+1} = l, \pi_{i+2}, ..., \pi_N|\pi_i = k)$
$= \sum_l e_l(x_{i+1})a_{kl} \sum_{\pi i+1...\pi N} P(x_{i+2}, ..., x_N, \pi_{i+2}, ..., \pi_N|\pi_{i+1} = l)$
$= \sum_l e_l(x_{i+1})a_{kl}b_l(i+1)$

Thus, we see that we can compute $b_k(i)$ for all k, i using dynamic programming. A summary of the backward algorithm follows:

- **Initialization:** $b_k(N) = 1$ for all k
- **Iteration:** $b_k(i) = \sum_l e_l(x_{i+1})a_{kl}b_l(i+1)$
- **Termination:** $P(x) = \sum_l a_{0l}e_l(x_1)b_l(1)$

Note a practical detail that to avoid dealing with small numbers, you can rescale at periodic positions by multiplying by a constant.

2.6.3 Forward and Backward Algorithm: Complexity

We now provide an analysis of the computational complexity of the forward and backward algorithms. Both algorithms run in $O(K^2N)$ time and $O(KN)$ space. We need a $K \times N$ sized matrix which yields the space complexity, and for each cell we need to do K work, which yields the time complexity.

2.6.4 Posterior Decoding

In posterior decoding, we are interested in finding the probability of being in state k at position i of the sequence x, or mathematically: $P(\pi_i = k|x)$. Since we have both the forward and backward probabilities, we see that:

$$P(\pi_i = k|x) = \frac{P(\pi_i=k,x)}{P(x)} = \frac{P(x_1...x_i,\pi_i=k)P(x_{i+1}...x_N|\pi=k)}{P(x)} = \frac{f_k(i)b_k(i)}{P(x)}$$

Thus, with posterior decoding we can find the most likely state at each position, a fact that is in general more helpful than the Viterbi path. It is also possible that posterior decoding might give an invalid sequence of states, in contrast to the Viterbi decoding. The Viterbi algorithm yields the most likely valid sequence of states that generated a given sequence x, while the posterior decoding gives the most likely state at each position despite the caveat that the resulting path may not necessarily be valid.

Posterior decoding will be covered more in detail next lecture, along with more investigation of HMMs.

3 References

Burrows, M. and Wheeler, D.J. A block-sorting lossless data compression algorithm. 1994.

Heng, L. and Durbin, R. Fast and accurate short read alignment with Burrows-Wheeler transform. 2009.

Langmead, B., Trapnell. C, Pop. M, Salzberg S. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome.

Rabiner, L. and Juang, B.H. An Introduction to Hidden Markov Models. 1986.