Advanced Algorithms

Class Notes for Thursday, November 29, 2012

Bernard Moret

1 Dynamic Programming: HMMs

Let $S = \{s_1, s_2, \ldots, s_n\}$ be the set of states of the HMM and let Σ be its output alphabet. Let the state transition matrix be the $n \times n$ stochastic matrix $A = (a_{ij})$, where a_{ij} is the probability of a transition from state i to state j, and let the emission matrix be the $n \times |\Sigma|$ matrix $E = (e_{ij})$, where e_{ij} is the probability of producing character j when in state s_i. Denote by $\alpha \cdot n$ the number of nonzero entries in matrix A. Finally, denote by x the observed output string, by x_i the character in the ith position of x, and by m the length of x.

We propose three related questions:

1. How do we reconstruct (and compute the likelihood of) the most likely state sequence for the given HMM and given output sequence?
2. How do we compute the likelihood of a given sequence?
3. How do we compute the likelihood of a particular state at a particular step for the observed output sequence?

All three of these questions can be answered by the same basic dynamic program, with small variations. We will build a table with one row for each state and one column for each position in the observed output sequence—thus our table will be $n \times m$. (Note that we may want to introduce two additional states, i.e., two additional rows, and one additional column; the extra states are a start state and an end state, the extra column corresponds to position 0—before the position of the first characters in the output string. These additions do not alter the following, but prevent having to treat special rows or columns differently from others and thus are likely to improve efficiency.)

1. The solution here is the so-called Viterbi algorithm. We will fill the table column by column, starting at column 0 and ending at column m; entry ij in the table, that is, $T(i, j)$, will denote the probability of the most likely path to state s_i after reading the jth character in x. We can compute $T(i, j)$ according to the following recurrence:

   $$T_v(i, j) = \max_l \left(a_{li} \cdot e_{i,x_j} \cdot T_v(l, j-1) \right)$$

   In words, we just look one step back along the path, to a previous entry $T_v(l, j-1)$ for some previous state l, and compute the probability of the extension from that previous step to the current state s_i at step j with symbol x_j emitted, retaining the largest one.

It should be noted, that, in spite of appearances, what is computed is really a joint probability, not (at least not intentionally) a conditional probability. That is, the recurrence computes the maximum value of the probability $P(\pi, x)$, where x is the output string and π is a path through the states of the HMM that produces that string. However, it is not hard to see that computing the max (over choices of π) of $P(\pi, x)$ is equivalent to computing the max of $P(\pi \mid x)$, the conditional (posterior) probability—the two are simply related by a factor of $p(x)$, which is fixed for a given x.

2. To get the likelihood of the output sequence, we could look at every state path that can generate it, compute the probability of each such path, then sum them all to obtain $P(x) = \sum_\pi P(\pi, x)$, using the notation from (i); but this takes time proportional to the number of paths and thus could take exponential time. Instead, we implicitly look at all paths every time we add another output character. The recurrence is nearly identical to the Viterbi recurrence—we simply replace the max operator by an addition.

$$T_f(i, j) = \sum_l \left(a_{li} \cdot e_{i,x_j} \cdot T_f(l, j-1)\right)$$

(Note that the emission probability is independent of the summation index and so can be factored.) The dynamic program resulting from this recurrence is often called the forward algorithm, hence my choice of T_f for this function.

3. Here we really want to compute $P(s_i, j \mid x)$, the probability that, given that the HMM produced the output sequence x, it was in state s_i at step j. Because this is a conditional probability (unlike the two computed above), we proceed somewhat indirectly, using joint probabilities that we do know how to compute. We have

$$P(s_i, j, x) = P(s_i, x_1 x_2 \cdots x_j) \cdot P(x_{j+1} x_{j+2} \cdots x_m \mid s_i, x_1 x_2 \cdots x_j)$$

Note that the first term in the product is exactly $T(i, j)$ in the forward algorithm, so we know how to compute it. The second term can be simplified: thanks to the Markov (memoryless) property, the dependency on $x_1 x_2 \cdots x_j$ does not exist, so the second term now reduces to $P(x_{j+1} x_{j+2} \cdots x_m \mid s_i)$, and we can easily compute that term by the same dynamic program again, but this time running it backward (which gives us the condition for free). The recurrence is then

$$T_b(i, j) = \sum_l a_{li} \cdot e_{i,x_{j+1}} \cdot T_b(l, j+1)$$

where the b subscript denotes that the backward version. Now we simply have

$$P(s_i, j \mid x) = \frac{T_f(i, j) \cdot T_b(i, j)}{P(x)}$$

What is the running time of each of these three algorithms? All have to fill in the entire $n \times m$ table; moreover, in order to fill it, all have to look at all transitions into (or out of) each state. Thus, to process one column of the table, all transitions of the HMM must be examined and used in the recurrence, so that the cost of processing one entire column is proportional to $\alpha \cdot n$, the number of nonzero entries in the transition matrix. Thus the overall running time of the algorithms is $\Theta(\alpha \cdot n \cdot m)$; this can be as low as order mn if α (which is twice the average degree of a node in the state transition diagram) is a constant and as high as order mn^2 if α is some fraction of n (as in a dense state graph).

2 Randomized Algorithms Redux

We return to randomized algorithms. We will begin with algorithms that build results incrementally and choose at random what part to build next. The best application of these algorithms is in computational geometry, but to illustrate the principle we first return to our old acquaintances quicksort and treesort. Both of course exist as deterministic algorithms, in which case, without modifications, both run in $O(n^2)$ time. Both can be protected against quadratic behavior in a deterministic way (quicksort through the very complex median-finding algorithm, treesort through any balanced tree), but both are typically used in their randomized version, which protects them against repeatable quadratic behavior while retaining their simplicity and speed. In their randomized versions, these sorting algorithms run in $\tilde{O}(n \log n)$ time, the tilde denoting expected running time. The similarity between the two algorithms runs deep: quicksort creates a tree of recursive calls and the better balanced this tree is, the faster quicksort runs; treesort builds a search tree that is a perfect image of the calling tree of quicksort and it too runs fastest when that tree is well balanced. However, the two differ in one major way: quicksort handles all of the data it is given immediately, before even making the first recursive call, since its first step is to partition the data. Indeed, the array in which it runs contains all of the data at any step—from step to step, the array simply becomes better sorted. In sharp contrast, treesort starts with a single item that it picks at random and creates a trivial tree of one node; as it progresses through steps, it slowly adds to the size of that tree while reducing the pool of unprocessed data. In that sense, quicksort is a bit like an iterative improvement algorithm, treesort more like a greedy algorithm: treesort proceeds through incremental construction. Incremental construction algorithms are typically easy to improve through randomization, although few are as simple as treesort.

The crucial attribute of an incremental construction algorithm is the separation of data into an unprocessed pile and a structured output. In treesort, we start with a set of values in some unspecified data structure; but we produce a binary search tree, a highly structure representation. The algorithm simply transfers a datum from the unprocessed pile to the structured output. Yet the input collection need not be purely unprocessed as in tree sort: randomized incremental construction algorithms typically do process the input to build a transient data structure that speeds up the transformation of input data into structured output. Unprocessed (or only partially processed) data are called conflict elements; if these data have been organized into some structure, they are then called conflict lists.

We illustrate the principle with the construction of 2D convex hulls. We start with an unprocessed collection of points, $P = \{p_1, p_2, \ldots, p_n\}$, and no output hull. We do a very simple processing of the input, using only linear time, to obtain a first very simple convex hull and, more importantly, a structured representation of the remaining points and how they relate to the current hull. Our first hull will just be a triangle, obtained by picking three points from the input at random; we will then eliminate any of the remaining points that fall within this triangle, using a single pass through all $n-3$ remaining points. (Testing for containment within a triangle is a $O(1)$ procedure.) Now surviving unprocessed points are all outside the hull, so that, when time comes to add one, it will become part of the next iteration of the hull and its addition will cause the removal of one or more edges of the current hull (as well as the addition of exactly two new edges to the hull). Each of these points is thus in conflict with some edges of the current hull—together, they form a *conflict set*.

We will set up and maintain a *conflict graph*, which indicates which remaining point is in conflict with which hull edge. This graph will be a bipartite graph, with one vertex for each element of the conflict set (each remaining unprocessed point) and one vertex for each edge of the current hull. In this bipartite graph, an edge connecting vertex p_j, representing an unprocessed point, to vertex e, an edge of the hull, indicates that v conflicts with e—i.e., that addition of v to the hull would cause removal of e. Geometrically, a vertex v outside the hull is in conflict with some edge $\{a,b\}$ of the current hull if and only if the triangle (a,b,v) intersects the hull only in the edge $\{a,b\}$—or, equivalently, if every point on the edge is "visible" to the vertex, with no intervening part of the current hull. (In practice, it suffices to test that the segments $\overline{va}$ and $\overline{vb}$ do not intersect the hull except at points a, respectively b, so that testing for conflict with a hull of 3 points takes constant time.)) Thus we can set up the initial conflict graph in linear time. Thereafter, our algorithm will select one of the remaining conflict points at random, determine the two new edges it will add to the hull, eliminate the edges with which it is in conflict, remove any conflict points that fall within the enlarged hull, and update the conflict graph. The cost of processing the next conflict point will thus vary hugely—but we will analyze this cost in probabilistic terms, not worst-case terms.

We first take a look at maintaining the hull and the set of conflict points. To maintain the hull, we need to identify edges to remove and also the vertices to which the new vertex will be connected. The first part is trivial thanks to the conflict graph: the adjacency list of the new vertex in the conflict graph points exactly to those edges that will need to be removed. The delete edges form a chain and it is precisely the endpoints of this chain that are the points to which the new vertex will be connected. If d is the degree in the conflict graph of the new vertex, then all of this can easily be carried out in $O(d)$ time. Next, we need to remove conflict vertices that fall within the enlarged hull. The new part of the hull is bounded by the two added edges and by the chain of d removed edges: thus points to be removed are conflict points that appeared on the adjacency list of one of the d removed edges and are inside the wedge determined by the two new edges. Testing for containment within the wedge is simply testing for containment within a polygon of two edges and takes constant time per point; thus we can simply take the adjacency lists of the d edges in conflict with the new vertex and filter them, which takes time proportional to the sum of the degrees in the conflict graph of the removed edges. It now remains to define how to update the conflict graph and then to analyze the expected running time of the entire algorithm.