Bioinformatics Coursework

gwwb64

January 2021

1 Question 1

1.1 Part a

For the MM: Finite alphabet Σ, Finite set of states S, initial probabilities $(p(s))_{s\in S}$, transition matrix $(m_{st})_{s,t\in S}$, emission probabilities $(e_s(a))_{s\in S,a\in\Sigma}$. For each state S, $\sum_{a\in\Sigma} e_s(a) \leq 1$ and $q(s) = 1 - \sum_{a\in\Sigma} e_s(a)$ where $q(s)$ is the probability of state s remaining silent. The sequence input is length n and the sequence will be denoted as A with A_j representing an observation in the sequence at time j. The assumptions state that the sum of all emission probabilities is less than or equal to 1 and that there is a 1 - sum of all emission probabilities that a given state is silent and does not emit a character.
Firstly, this model must be transformed to create distinct silent states instead of using emission probabilities for silent states.
1. For every emission probability of every state, divide the probability by the sum of the emission probabilities for that state and assign this as the new emission probability. Each state should have its ability to remain silent removed at this point in implementations.

$$\forall s \in S, \forall a \in \Sigma. \quad g_s(a) = \frac{e_s(a)}{\sum_{a\in\Sigma} e_s(a)}$$

2. For every state in the model, create a new state which will be silent (this will double the number of states). For the purposes of notation, silent states corresponding to non-silent states will be denoted with a subscript q i.e. the silent state corresponding to state s will be denoted s_q.
3. For every state transition in the transition matrix, multiply the transition probability by the probability of the end state not being silent.

$$\forall s, t \in S. \quad r_{st} = m_{st} \sum_{a\in\Sigma} e_t(a)$$

4. For every state transition, add a new state transition corresponding to the added silent states from step 2. The probability for this new transition should be the probability of the original transition from the previous state to the original non-silent state in the MM definition multiplied by the probability of the end state remaining silent from the original definition.

$$\forall s, t \in S. \quad r_{st_q} = m_{st}(1 - \sum_{a\in\Sigma} e_t(a))$$

Let the state sequence be a path π. The most probable path can be defined by

$$\pi^* = \text{argmax}_\pi P(A, \pi)$$

Now apply a modified version of the Viterbi algorithm.
1. Set up a Dynamic Programming table to store Viterbi probabilities. Rows will be states and columns will be time steps. 2. Using the DP table: Let $v_k(i)$ be the probability of the most probable path over the first i characters and ending in state k. Initialize the most probable path at time t = 0 for each state using the initial probabilities

$$\forall k \in S. \quad v_k(0) = p(k)e_s(A_0).$$

3. Recursion (i = 1 ... L)
Handle non-silent states:

$$v_s(i) = e_s(A_i)\text{max}_k(v_k(i-1)r_{ks})$$

$$\text{ptr}_i(s) = \text{argmax}_k(v_k(i-1)r_{ks})$$

Handle silent states:

$$v_{s_q}(i) = \max_k(v_k(i)r_{ks})$$

$$\text{ptr}_i(s_q) = \text{argmax}_k(v_k(i)r_{ks})$$

In topological order of silent states, take the silent state s_q

$$v_{s_q}(i) = v_{s_q}(i) + \max_k(v_k(i)r_{ks_q})$$

for all silent states where $k < s_q$.

4. Termination (end states are assumed)

$$P(A, \pi^*) = \max_k(v_k(L)r_{k0})$$

$$\pi^*_L = \text{argmax}_k(v_k(L)r_{k0})$$

5. Traceback (i = L ... 1)

$$\pi^*_{i-1} = \text{ptr}_i(\pi^*_i)$$

Proof of correctness for our method of adding silent states to the Markov Model: For the purposes of this proof: S is a non-silent state, t is the probability of a transition between 2 states, D is a silent state, R is a combined pseudo-state of the original state S and the added state D. $e_s(a)$ is the emission probability in state s of character a as in the original MM definition. Using Figure 1

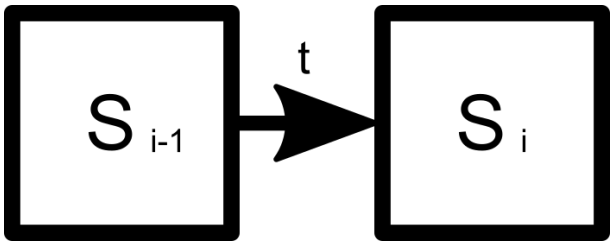

Figure 1: Diagram of original state transitions

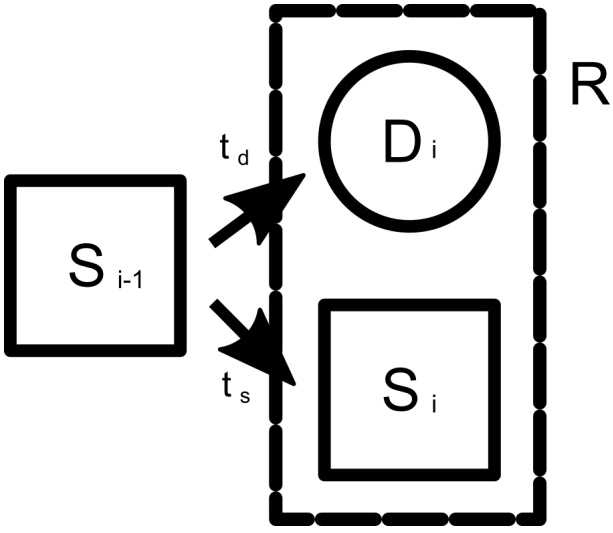

Figure 2: Diagram of modified state transitions

$$\sum_{a \in \Sigma} e_{S_i}(a) + (1 - \sum_{a \in \Sigma} e_{S_i}(a)) = 1$$

which is what would be expected since these are not independent variables. Therefore the combined independent transition and (emission + silent) probabilities $= t \cdot 1 = t$
Using Figure 2: Applying the emission probability transformation equation

$$\forall a \in \Sigma. \quad g_{S_i}(a) = \frac{e_{S_i}(a)}{\sum_{a \in \Sigma} e_{S_i}(a)}$$

means that

$$\sum_{a \in \Sigma} g_{S_i}(a) = 1$$

As mentioned we can redefine the transition probabilities so

$$t_d = t \sum_{a \in \Sigma} e_{S_i}(a)$$

and

$$t_s = t(1 - \sum_{a \in \Sigma} e_{S_i}(a))$$

The silent state D_i emits nothing but for the purposes of this proof we can assume it has an emission probability of 1 for a null pseudo-character $e_{D_i}(null) = 1$. Since Hidden Markov Models assume independence between state transitions and emissions, we can use this fact to prove that our pseudo-state R makes the overall model the same.

$$P(R \text{ is silent}) = t_d \cdot e_{D_i}(null)$$

$$P(R \text{ is not silent}) = t_s \cdot \sum_{a \in \Sigma} g_{S_i}(a)$$

$$t_d + t_s = t$$

$P(R \text{ is silent}) + P(R \text{ is not silent}) = 1 \cdot t$ since these are not independent and represent the possible actions within R including state transitions.

$$t \sum_{a \in \Sigma} e_{S_i}(a) \cdot e_{D_i}(null) + t(1 - \sum_{a \in \Sigma} e_{S_i}(a)) \cdot \sum_{a \in \Sigma} g_{S_i}(a) =$$

$$t \sum_{a \in \Sigma} e_{S_i}(a) \cdot e_{D_i}(null) + t - t \sum_{a \in \Sigma} e_{S_i}(a) \cdot \sum_{a \in \Sigma} g_{S_i}(a) =$$

$$t \sum_{a \in \Sigma} e_{S_i}(a) \cdot 1 + t - t \sum_{a \in \Sigma} e_{S_i}(a) \cdot 1 =$$

$$t$$

We have shown that the combined independent transition probability and (emission + silent) probabilities remain the same after splitting the model into a silent + non-silent state.
Proof of correctness for general Viterbi:
As mentioned prior to the Viterbi explanation, we are trying to find the most likely series of states $\pi^* = \pi_1, ..., \pi_n$ for the observed sequence $A = A_1, ..., A_n$. We will start with the assumption

$$V_k(i) = \max_{\pi_1, ..., \pi_{i-1}} \cdot P(A_1, ..., A_{i-1}, \pi_1, ..., \pi_{i-1}, A_i, \pi_i = k)$$

which represents the probability of the most likely sequence of states having a final state of k.

$$V_s(i+1) = \max_{\pi_1, ..., \pi_i} \cdot P(A_1, ..., A_i, \pi_1, ..., \pi_i, A_{i+1}, \pi_{i+1} = s)$$

$$= \max_{\pi_1, ..., \pi_i} \cdot P(A_{i+1}, \pi_{i+1} = s | A_1, ..., A_i, \pi_1, ..., \pi_i) \cdot P(A_1, ..., A_i, \pi_1, ..., \pi_i)$$

$$= \max_{\pi_1, ..., \pi_i} \cdot P(A_{i+1}, \pi_{i+1} = s | \pi_i) \cdot P(A_1, ..., A_i, \pi_1, ..., \pi_i)$$

$$= \max_k [P(A_{i+1}, \pi_{i+1} = s | \pi_i = k)] \cdot \max_{\pi_1, ..., \pi_{i-1}} [P(A_1, ..., A_{i-1}, \pi_1, ..., \pi_{i-1}, A_i, \pi_i = k)]$$

$$= \max_k[P(A_{i+1}, \pi_{i+1} = s) \cdot P(\pi_{i+1} = s|\pi_i = k) \cdot V_k(i)]$$
$$= e_s(A_i)\max_k[r_{ks}]$$

The modification for silent states involves changing the method slightly to first calculate V for the final non-silent state for the partial sequence, then adding the sum of Vs for topologically prior silent states.
The space complexity of the regular Viterbi algorithm is $O(kn)$ since we need to store a $k \cdot n$ matrix and the time complexity is $O(k^2n)$. With our modifications, the number of states is doubled so technically the space complexity will be $O(2kn)$ and the time complexity will be $O(2k^2n)$ but for sufficiently large n or k, the constant 2 will be negligible.

1.2 Part b

The EM algorithm for this HMM is the Baum-Welch algorithm. The algorithm starts with a set of model parameters: the initial probabilities, transition probabilities and emissions probabilities. These can be random. The forward and backward algorithms are then run which are dynamic programming algorithms. The forward algorithm determines the probability of a state at a certain time, given past states and observations while the backward algorithm calculates the probability of the remaining observations for any start time. The probability of the full sequence, P is also determined by the forward algorithm through summation of all possible state path probabilities. The forward algorithm computes the emission probability * sum of products of previous calculated probability and transition probability. The backward algorithm does the same except the emission probability is included in the summed products. After the forward/backward computes the posterior marginal probabilities, variables A and E are initialised which represent the expected number of times transitions and emissions are used respectively given the observed sequence and parameters. A values are calculated from forward, backward, emission and transition probabilities multiplied and summed, then divided by P. E values are calculated from the forward * backward probabilities summed and divided by P. Initial probabilities can just be updated from the first row of E. Transition probabilities are calculated, from the sum for all time steps, of each A value divided by each E value. Emission probabilities are calculated from the sum of E values where the observed character is found divided by the sum of all E values for all time steps. Empirical tests can be done by measuring the log likelihood of the forward algorithm. Over many iterations (or sets of sequences), the log-likelihood should converge if training is successful. See Figures 3 - 12. There were 2 GitHub repositories used for reference in learning about how implementations of Baum-Welch usually work. See [3][2].

```
hfac = HMMFactory()
m = hfac.random_model(2, ['A', 'B', 'C'])
l = m.train_baum_welch('ABBCBACBCBABCBCBABC', 500)
plt.plot(l)
plt.show()
```

Figure 3: Test1 Settings

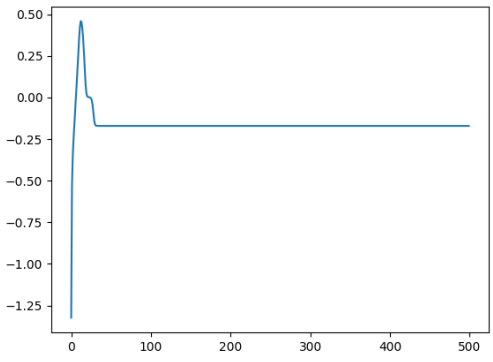

Figure 4: Test1 Results

```
hfac = HMMFactory()
m = hfac.random_model(2, ['A', 'B', 'C'])
l = m.train_baum_welch('AAAAAAAAAA', 500)
plt.plot(l)
plt.show()
```

Figure 5: Test2 Settings

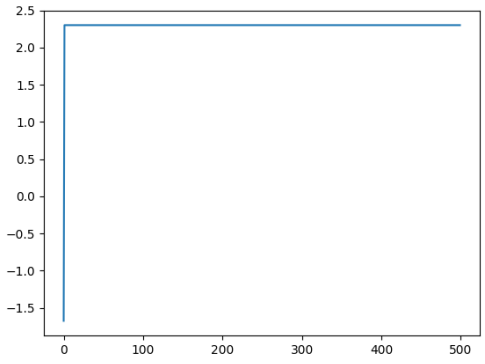

Figure 6: Test2 Results

```
hfac = HMMFactory()
m = hfac.random_model(2, ['A', 'B', 'C'])
m2 = hfac.random_model(2, ['A', 'B', 'C'])
seq = m2.generate(100)
print(seq)
l = m.train_baum_welch(seq, 500)
plt.plot(l)
plt.show()
```

Figure 7: Test3 Settings

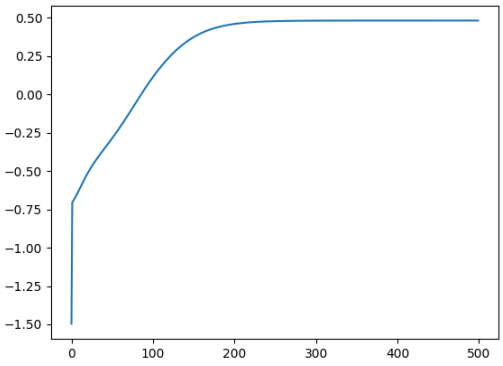

Figure 8: Test3 Results

```
hfac = HMMFactory()
m = hfac.random_model(5, ['A', 'B', 'C'])
m2 = hfac.random_model(5, ['A', 'B', 'C'])
seq = m2.generate(500)
print(seq)
l = m.train_baum_welch(seq, 500)
plt.plot(l)
plt.show()
```

Figure 9: Test4 Settings

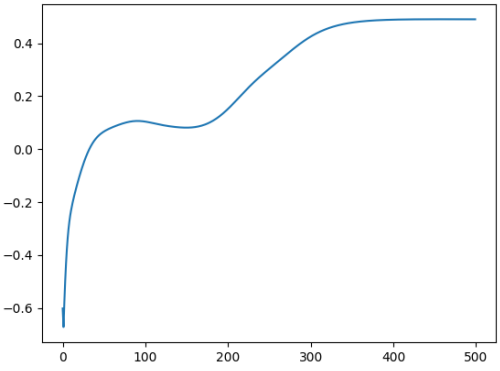

Figure 10: Test4 Results

BCCACBCBACACACBCACACABCCBACBBACBCBBBBBCBBACCBCACBBACAAACBCBACACBACCACACAABCACABA
BCCCACBAACCBBBAABACC

Figure 11: Test3 Sequence

BCBACCAAAAAAACBBBCBBAACAACCCBCBBCCBABBAABBCCBBCBCBCAACBCBAAAACACABBCCABCBBCACABA
ABBAABCBCCCAACAABAAABCBABBBABBCBBABACCABCACACBABCCCACABAABCBBAABACBBCAABACBCACCBC
ABAABABBCCACCAACCCACCCBCBABCAAAAABAAACCBCBCBABCCCBBBAACBCBABBBCCBAACABBABCACCACB
ABBCCACACCACACABBBAAABACACCBABBBACAABCCAACBCACBCABCCBABBCCABAAABCBAACACCAABCBAAA
CAABBCACABAAAABBBCACCCBBCCABBCAABACBABCABCCACCBABAAABAACAABCBCBABCAABBACCACCCAAA
ABCCCBBCBCBBCBBCACABAAAACCACBBCAACCABCCCBCABBBAABBACACBBBCBAACBCABBBBBCABCCACACC
CAABCCCACBACBCABAACC

Figure 12: Test4 Sequence

2 Question 2

2.1 Part a

The BUILD [1] algorithm creates a tree from a set of constraints representing the relationships between descendants and common ancestors. The algorithm requires a set of nodes and a set of constraints as

inputs. A tree containing only 1 node will be returned and the algorithm terminates if the input set of nodes contains only one item. The algorithm first computes blocks for each node i.e determines all the nodes at a given level with a common ancestor. If there is only one block returned by the block computation then the currently being processed node has only one child worth of leaf nodes so the null tree is returned. For every block calculated, a subset of constraints is selected to ensure that the only constraints being processed in the current BUILD invocation are ones relating to nodes in the current block. The BUILD algorithm is recursively called for the subset of nodes in the calculated block and the associated constraints. Each recursive call can of course return the null tree at this stage in which case this is passed up the call stack. After each block has been processed, if the null tree wasn't returned then it means that a tree must exist for a set of constraints and a new tree is returned with a root node and each internal node is a root for the leaf nodes in the calculated blocks.

2.2 Part b

function PARTITION(C)
Initialize a data structure with keys and values such that keys represent leaves and values represent block ids, B. Initialize all values to null at first
Initialize a variable for the number of blocks, $n = 1$
for constraint in C **do**
Let i, j = left side of the constraint
if i or j but not both already have non-null entries in B **then**
Assign the one without an entry to have the same block value in B as the found entry
else if neither have a non-null entry in B **then**
Assign both to have a value in B equal to n i.e. $B[entry] = n$
Increment n
else if both already have a unique non-null entry in B **then**
Merge the 2 blocks by assigning all relevant entries with the first value
end if
end for
for constraint in C **do**
Let i, j = left side of the constraint
Let k, l = right side of the constraint
if B[k] = B[l] **then**
B[k] = B[i]
B[l] = B[i]
end if
end for
Let A be a list of blocks where each block is a list of nodes
for entry in B **do**
Let k, v = entry
Append k to A[v]
end for
Adjust the keys in A such that they are in range 1 to m where m is the number of unique keys. This is necessary since the algorithm may have merged entire blocks creating holes in the count.
Add any remaining nodes that had no constraints (i.e. have a null value in B) as new blocks
return A
end function

2.3 Part c

Partition recursion 1
Block 1: chajnlef
Block 2: digb
Block 3: km

Partition recursion 2
Block 1: cha
Block 2: jnl
Block 3: ef

Partition recursion 3
Block 1: ch
Block 2: a

Partition recursion 4
Block 1: c
Block 2: h

Partition recursion 5
Block 1: c

Partition recursion 6
Block 1: h

Partition recursion 7
Block 1: a

Partition recursion 8
Block 1: jn
Block 2: l

Partition recursion 9
Block 1: j
Block 2: n

Partition recursion 10
Block 1: j

Partition recursion 11
Block 1: n

Partition recursion 12
Block 1: l

Partition recursion 13
Block 1: e
Block 2: f

Partition recursion 14
Block 1: e

Partition recursion 15
Block 1: f

Partition recursion 16
Block 1: di
Block 2: gb

Partition recursion 17
Block 1: d
Block 2: i

Partition recursion 18
Block 1: d

Partition recursion 19
Block 1: i

Partition recursion 20
Block 1: g
Block 2: b

Partition recursion 21
Block 1: g

Partition recursion 22
Block 1: b

Partition recursion 23
Block 1: k
Block 2: m

Partition recursion 24
Block 1: k

Partition recursion 25
Block 1: m

2.4 Part d

For the purposes of this algorithm, we consider a block to be all leaf nodes connected to a common node.

function UNBUILD(T)
Perform a Depth-First-Search on T, recording each path to a leaf in P
Initialize a set of candidate constraints C
for Node i in T **do**
for Node j in T **do**
if $i == j$ **then**
Skip this iteration
end if
for Node k in T **do**
for Node l in T **do**
if $k == l$ **then**
Skip this iteration
end if
Store candidate constraint $(i, j) < (k, l)$ in a temporary variable c
if Lowest common ancestor of (i, j) is a proper descendant of the lowest common ancestor of (k, l) by checking against respective paths in P **then**
if C does not already contain a constraint with an $(i_c, j_c) < (k_c, l_c)$ such that

the 4 blocks mentioned in the current candidate constraint match that of an existing constraint (both blocks on LHS same between C and candidate and both blocks on RHS same between C and candidate) in C or any leaf in i, j, k, l has not yet been given a constraint in C **then**
Store candidate constraint c into set C
end if
end if
end for
end for
end for
end for
end function

The search space of this algorithm could be reduced by pre-generating the (i, j) pairs from the leaves based on the blocks and only searching for (k, l) pairs. However, care must be taken in doing this since the left hand side of a constraint can sometimes require nodes that are not in the same block of the final tree from BUILD. See Fig 3 from [1]. The proof of this algorithm is very simple. We can prove the base case with 3 nodes where one is a direct descendant of the root and the other 2 are descendants of a descendant node of the root. In this case, 1 constraint $(1, 2) < (1, 3)$ will be generated. Variants of this such as $(2, 1) < (3, 1)$ will be rejected since the algorithm will see that a constraint with blocks 1 & 2 already exists and there are no leaves out of the candidate $(2, 1) < (3, 1)$ that don't appear in a constraint in C. Any combination of leaves that don't match the correct ancestry constraint are also rejected. If we assume that the algorithm is correct for a k leaf tree, it will also be correct for a tree of $k + 1$ leaves since in any case the new leaf extends the block size of the ancestor it is attached to. At least one constraint must be added to satisfy the requirement that all nodes are mentioned in a constraint.

References

[1] A. Aho et al. "Inferring a Tree from Lowest Common Ancestors with an Application to the Optimization of Relational Expressions". In: *SIAM J. Comput.* 10 (1981), pp. 405–421.

[2] Abhisek Jana. *HiddenMarkovModel*. 2019. URL: `https://github.com/adeveloperdiary/HiddenMarkovModel/blob/master/part3/BaumWelch.py`.

[3] Hamza Rawal. *HMM-Baum-Welch-Algorithm*. 2019. URL: `https://github.com/hamzarawal/HMM-Baum-Welch-Algorithm/blob/master/baum-welch.py`.