Kallisto Project

Matthew Burke, Jansel Herrera, Modeste Astague

May 2019

For our final project, we implemented a simplified version of the popular software program kallisto. Kallisto is a program for quantifying abundances of transcripts from RNA-seq data. Given a set of transcripts T and a set of reads R (both RNA sequences in fasta format), our program will output the relative abundance of each transcript $t \in T$ that is contained in R.

We originally intended to implement a simplified version of kallisto, and then further improve the implementation by speeding up the EM algorithm step to be even faster than kallisto itself. However, just implementing the simplified version of kallisto was such a large task that we did not have time to improve the EM algorithm implementation. The production kallisto code is too long and complicated (and in C++, a language none of us know) to understand how it works in a timely manner. We implemented our program based on our understanding of the end goal and what we thought was most efficient. We revised the structures we used several times to make our program as efficient as possible.

It is easiest to explain how our program works by explaining each method sequentially.

First the fasta files are converted to binary files.

Then, we read in the transcripts $t \in T$. We create an integer label for each transcript. Each transcript t contains [(length of t) - k] kmers. Each kmer is stored in hashTable. We implement a hash function on the kmer to find its row in hashTable, and then store a kmer-labelled struct at the first empty spot in the row. If a kmer is already contained in hashTable, we only add the transcript label to the kmer-labelled struct. The murmurhash3 function that we implement is the same hash function that kallisto uses. We found it to be much faster than the CMPH hashing program that we used in class.

Once all of the transcripts are read, the hashTable contains kmer-labelled structs with the kmer sequence and the transcript that the kmer came from. The from-reads attribute is zero for all of these structs, since no reads $r \in R$ have been read yet.

Next we read all of the reads $r \in R$. We want to match each read with a set of transcripts that it could have come from. We simplify this task to a simpler task. We want to match each read to a kmer in hashTable. This is much simpler than the deBrujin graph approach that kallisto uses. We let the first kmer of each read represent the read, and find a matching kmer in hashTable. For the matching kmer-labelled struct in hashTable, we increment from-reads by one. After all of the reads $r \in R$ have been read, hashTable contains the number of times that each kmer appeared at the beginning of a read.

For the EM algorithm, we do not need the count of each kmer. We only need the total count of all kmers in the same equivalence class. An equivalence class is the set of all transcripts that the kmer could have come from. We already determined the equivalence class of each kmer when we stored them in the hashTable. Kmer-labelled.t-labels represents the equivalence class of each kmer.

We create an array, eqc-arr, to store the equivalence class count c_e of each equivalence class contained in the reads. The total number of equivalence classes possible with n transcripts is 2^n. However, it is unlikely that every equivalence class will be observed, so we allocate less memory than that for eqc_arr. Each index in eqc-arr corresponds to a unique equivalence class. To store the equivalence class counts, we sum up the from-reads count of each kmer with the same equivalence class. The fastest way to do this is to loop through hashTable, and for every kmer that appeared in a read (from-reads ¿ 0), we add from-reads to the index of eqc-arr that corresponds to the equivalence class of the kmer.

Once we have the equivalence class counts, we no longer need hashTable. We free up the memory used by hashTable.

The expectation maximization algorithm variant from the kallisto paper is used to estimate the relative abundance of each transcript in the reads R. The EM algorithm utilizes the likelihood function

$$L(\alpha) \propto \prod_{f \in F} \sum_{t \in T} \mathbf{y}_{f,t} \frac{\alpha_t}{l_t} = \prod_{e \in E} \left(\sum_{t \in e} \frac{\alpha_t}{l_t} \right)^{c_e}$$

Where l_t is the lengths of the transcripts, which we stored in an array when we read in the transcripts,
c_e is the equivalence class counts, which are stored in eqc-arr,
e is the set of transcripts in each equivalency class, which we also stored in eqc-arr, and
α_t is the probability that a fragment f came from transcript t, the normalized abundance of t in R.

Our program ultimately outputs ρ_t, the abundance of each transcript, alongside the corresponding transcript names.

We transformed this likelihood function into the log-likelihood function

$$l(\alpha) \propto \sum_{e \in E} \left(c_e \cdot log(\sum_{t \in e} \frac{\alpha_t}{l_t}) \right)$$

The EM algorithm we implemented iterates through two steps, an E step and an M step, until the value of $l(\alpha)$ converges.

For notation, we use t_k to indicate the k^{th} transcript of K total transcripts. We use ρ to indicate the abundance of each transcript, and l to indicate the length of each transcript. S_i is the i^{th} equivalence class of a total of N equivalence classes.

Before we begin iterating through the EM algorithm, we initialize all values of ρ as $1/K$.

For the E step, we estimate values of the function f at iteration m with

$$f_{ik}^{(m)} = \begin{cases} \frac{\rho_k^{(m)}}{\sum_{j \in S_i} \rho_j^{(m)}} & \text{if } k \in S_i \\ 0 & \text{otherwise} \end{cases}$$

For the M step, we recompute the values of α and ρ with

$$\hat{\alpha}_k^{(m+1)} = \frac{1}{N} \sum_{i=1}^{N} f_{ik}^{(m)}$$

and

$$\hat{\rho}_k^{(m+1)} = \frac{\frac{\alpha_k^{(m+1)}}{\ell_k}}{\sum_{j=1}^{K} \frac{a_j^{(m+1)}}{\ell_j}}$$

Updating α and ρ in this manner is equivalent to computing

$$\alpha = argmax_{\alpha} \mathbb{E}[\sum_{e \in E} \left(c_e \cdot \sum_{t \in e} \frac{\alpha_t}{l_t} \right)]$$

but is more computationally efficient.

Our program outputs our predicted transcript abundance values ρ next to the true abundance values from the simulated data. Since we simplified the problem of matching a read to the transcripts it could have come from to only matching the first kmer of a read to transcripts it could have come from, our predictions are not very accurate. Our program also outputs the mean squared error of our predictions.

Our code is on gitlab in the folder janselh/JMMS.

Works Cited:

Bray, Nicolas L., et al. "Near-optimal probabilistic RNA-seq quantification." Nature biotechnology 34.5 (2016): 525.

Spear, Logan. "Lecture 12: RNA-seq -A Counting Problem." http://data-science-sequencing.github.io/Win2018/lectures/lecture12/algo2.