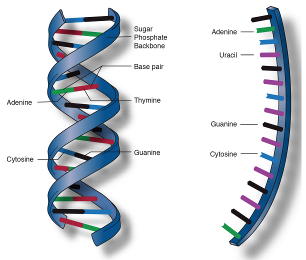
Data Parallel EM for estimating the Genome Relative Abundance (GRA) in Metagenomic Samples

Orren Karniol-Tambour

Setting: We've taken a sample from a microbial community - e.g. water from a pond, blood sample from a sick human. The sample contains traces of the DNA and RNA of viruses and bacteria living in the pond/body.



Deoxyribonucleic acid (DNA) Ribonucleic acid (RNA)

We perform shotgun sequencing on the sample and get a series of genomic reads - i.e. strings of nucleotide bases:

ACGTCGATCGCTAGCCGCATCAGCAAACAACACGCTACAGCCT

So we have:

- a set of known reference genomes (long strings).

- a set of reads (shorter strings), along with the number of high quality 'hits' from each read to each genome

(where a 'hit' reflects edit distance between the read string and substring of a reference genome below some threshold)

Our goal is to estimate the relative abundance of all known bacteria and viruses in the environment we sampled from - e.g. figure out why our patient is sick

We assume our reads are drawn iid from a mixture of genomes - so we can view the Genome Relative Abundance (GRA) as a finite mixture we need to estimate and use EM to solve:

Repeat until convergence: {

(E-step) For each i, j, set

$$w_j^{(i)} := p(z^{(i)} = j \mid x^{(i)}; \phi)$$

(M-step) Update the parameters:

$$\phi_j := \frac{1}{m} \sum_{i=1}^{m} w_j^{(i)}$$

}

EM - quick review

-iterative algorithm for finding maximum likelihood estimate of parameters when model depends on latent variables

-'missing' Z data matrix, where Zij tells us whether sample i came from source j

-pick a guess for parameters, estimate posterior distribution of the Zs given data X and current guess for parameters

-update parameters based on current guess for Zs

-improves on each iteration, converges to local optimum

EM applied to GRA estimation:

Key insight: we can approximate the likelihood of the data as # hits from read i on genome j, normalized by length of genome j (since hits on shorter genomes are more informative)

E-step

$$Z_{ij}^{(t)} = \frac{p(r_i \mid Z_{ij} = 1; G)\, \pi_j^{(t)}}{\sum_{k=1}^{n} p(r_i \mid Z_{ik} = 1; G)\, \pi_k^{(t)}} \approx \frac{(S_{ij} / L_j)\, \pi_j^{(t)}}{\sum_{k=1}^{n} (S_{ik} / L_k)\, \pi_k^{(t)}}$$

M-step

$$\pi_j^{(t+1)} = \frac{1}{m} \sum_{i=1}^{m} z_{ij}^{(t)}$$

Where:

r_i is the i'th read

Sij is the number of 'hits' from read i to genome j

L_j is the length of genome j

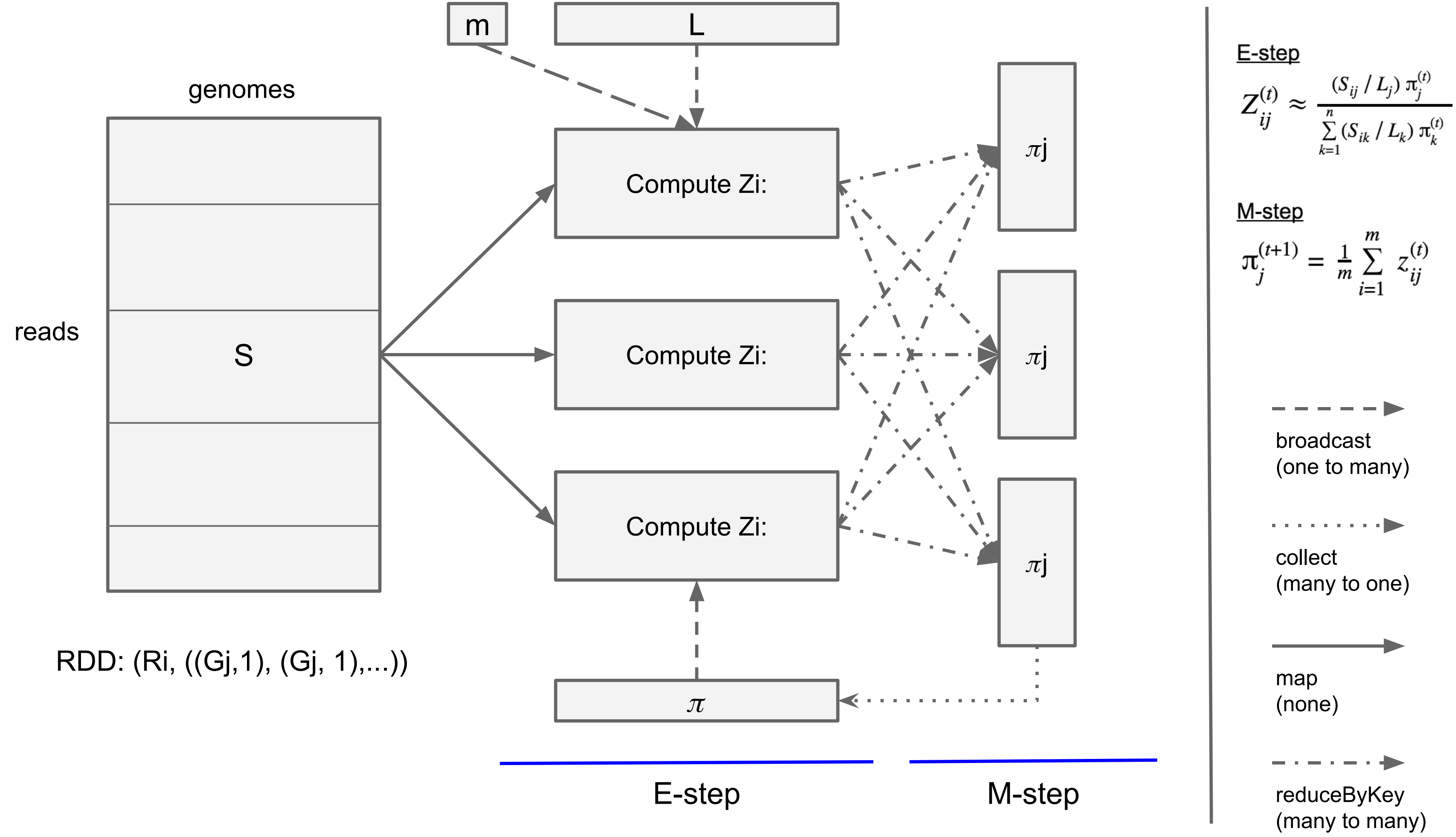
π_j is a mixing parameter that describes the contribution of the j'th genome to the mixture, and $\sum_{j=1}^{m} \pi_j = 1$

Xia et al., PLoS One 2011

Each iteration costs O(mn) time, where m is the number of reads, n is number of genomes

In practice, m is very large (millions) and getting larger as sequencing gets exponentially cheaper and 'deep' sequencing becomes common

n is manageable (thousands) and will grow far more slowly


m
L
genomes
reads
S
RDD: (Ri, ((Gj,1), (Gj, 1),...))
Compute Zi:
Compute Zi:
Compute Zi:
πj
πj
πj
π
E-step
M-step
E-step
$Z_{ij}^{(t)} \approx \frac{(S_{ij} / L_j)\, \pi_j^{(t)}}{\sum_{k=1}^{n} (S_{ik} / L_k)\, \pi_k^{(t)}}$
M-step
$\pi_j^{(t+1)} = \frac{1}{m} \sum_{i=1}^{m} z_{ij}^{(t)}$
broadcast
(one to many)
collect
(many to one)
map
(none)
reduceByKey
(many to many)

E-step

```
map(i, Si:) :
    n = length(Si:)
    sum = 0
    for j in n:
        nnZij = (Sij / Lj) Pi(j)
        sum += nnZij
    for j in n:
        nnZij = (Sij / Lj) Pi(j)
        Zij = nnZij / sum
        emit(j, Zij)
```

M-step

```
reduce(j, Z:j) :
    Pi(j) = sum(Z:j) / m
    emit(j, Pi(j))
```

Single Machine - Cost of Single Iteration

O(mn) time

Data Parallel EM - Cost of Single Iteration

Time

E-step: O(mn/B)

M-step: O(n/B)

Total: O(mn/B) time

embarrassingly parallel!

Communication

broadcast: O(nB)

shuffle: O(nB)

(with combiners)

Total: O(nB)

```
// -------- Initialize Pi --------

// get number of genomes
val numGenomes = lengths.value.size

// for now let's just make pi uniform.
var currentPi = lengths.value.keys.toList.map(r => (r, 1 / numGenomes.toDouble)).toMap
var newPi = currentPi

// create empty list to account for genomes we haven't seen
val emptyPi = lengths.value.keys.toList.map(r => (r.toInt, 0.0)).toList


// -------- Run EM Till Convergence --------

// params
val maxIterations = 1000
val convergenceTol = .000001
var iteration = 0
var maxdiff = 100

while (iteration <= maxIterations && maxdiff > convergenceTol) {


    // broadcast current pi Map to workers
    val pi = sc.broadcast(currentPi)

    // helper function, gets pi for a genome by key
    val getPi = (x: Int) => pi.value.get(x.toString).get.toDouble


    // -------  E step  -------

    // compute Zij
    val computeZ = (r: (String, List[(Int, Double)])) => {

        // non-normalized Zij
        val znn = r._2.map(x => (x._1, x._2 * getPi(x._1.toInt)))

        // sum of Zi: row
        val znnsum = znn.map(x => x._2).sum

        // normalized Zij
        val zn = znn.map(x => (x._1, x._2 / znnsum))

        // output (read-i, List((G1, Zi1), (G2, Zi2), ...))
        (r._1, zn)
    }

    // map iterator vals to List, and compute Zij's -- see format above
    val zmatrix = smatrix.mapValues(_.toList).map{r => computeZ(r)}

    // -------  M step  -------

    // compute new estimate of pi
    val piNew  = zmatrix.flatMap(x => x._2)   // flatmap Z to get (Gj, Zij) tuples

        // reduce to sum, map to divide, getting (Gj, PIj) tuples
        // this takes an avg over the Z:j column
        .reduceByKey(_ + _).map(x => (x._1, x._2/ numReads))

        // collect to driver as list
        .collect().toList


    // merge new and empty pi lists to get new pi
    newPi = (emptyPi ++ piNew).groupBy(_._1)
        .map(kv => (kv._1.toString, kv._2.map(_._2).sum))


    // -------  Calculate Residual -------

    // take max abs pairwise diff of pi new-old, equivalent to GRAMMy's maxd() c++ function
    val diffPi = (newPi ++ currentPi).groupBy(_._1)
        .map(kv => (kv._1, kv._2.map(_._2)
        .reduce(_ - _))).toList
    var maxdiff = scala.math.abs(diffPi.maxBy(x => scala.math.abs(x._2))._2)

    // assign new pi to current
    currentPi = newPi


    iteration += 1
}
```