Supplemental File 1

August 5, 2020

1 The Sequential CoGAPS Algorithm

A description of the sampling steps employed by the serial version of CoGAPS is essential to describe the modifications necessary to enable parallelization. Briefly, CoGAPS is a sparse Bayesian NMF. The input for CoGAPS is a data matrix, $D \in \mathbb{R}^{N \times M}$, an uncertainty matrix, $\Sigma \in \mathbb{R}_+^{N \times M}$, and a number of patterns (latent spaces) to learn, K. It factors D into two lower dimensional matrices, $A \in \mathbb{R}_+^{N \times K}$ and $P \in \mathbb{R}_+^{M \times K}$. The prior distribution of A is given by

$$A_{i,k} \sim \Gamma\left(\alpha_{i,k}^A, \lambda\right) \tag{1}$$

and the prior distribution of P is

$$P_{j,k} \sim \Gamma\left(\alpha_{j,k}^P, \lambda\right), \tag{2}$$

where both $\alpha_{i,k}^A$ and $\alpha_{j,k}^P$ have a Poisson prior with parameter α. The likelihood of the data in CoGAPS is given by

$$D \sim \mathcal{N}(AP^T, \Sigma). \tag{3}$$

In this model, the values of α and λ control the relative sparsity of the A and P matrices. The parameter, λ, is set as a function of α so that the level of sparsity is inversely proportional to the expected size of each matrix element to better fit the data:

$$\lambda = \alpha \sqrt{\frac{K}{\overline{\overline{D}}}} \tag{4}$$

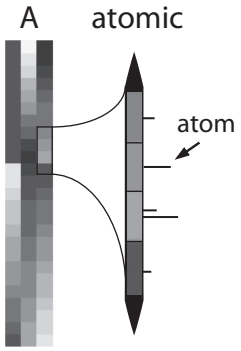

Figure 1: Illustration of the atomic domain for a sample matrix A. The values of A are depicted in a heatmap in a grayscale where black corresponds to a value of zero (left). The elements $A_{8,3}$ to $A_{11,3}$ are zoomed in with a depiction of the corresponding bins in $\mathcal{A}$, the atomic domain for A. Each atom has a position along this domain l indicated by the height of the horizontal line segment and magnitude x indicated by the length of that line segment. The atomic domain is divided into bins so that the values l determine to which matrix element each atom maps. The value of each matrix element is determined by the sum of the magnitudes x for all atoms mapping to that element. In this example, $A_{8,3}$, $A_{9,3}$, and $A_{11,3}$ all contain a single atom and whereas $A_{10,3}$ contains two atoms. Elements containing no atoms would be exactly zero.

Rather than working with values directly stored in the matrix elements, CoGAPS samples from an atomic prior [1]. This prior is represented by a continuous domain of point masses. Each point mass is called an

atom and has a position and mass within this continuous domain (Fig 1). The atomic domain is separated into $N \cdot K$ bins for the A matrix and $M \cdot K$ bins for the P matrix. In this way, each position in the atomic domain maps to a single element of the corresponding matrix. The value of a particular matrix element is equal to the sum of all atom masses in the corresponding bin of the atomic domain or is exactly zero if there are no atoms in the region of the domain corresponding to that matrix element. We let the mass of each atom be distributed according to an exponential prior with parameter λ. Because a Gamma distribution is the sum of exponentials, this atomic prior naturally models the Gamma prior imposed on each matrix element. Performing update steps that work directly on the position and mass of the atoms instead of the matrix elements provides a natural way to model the Poisson prior on the shape of the Gamma distribution for each matrix element. It also allows for exact zeros that could not be readily performed sampling from the matrix elements themselves. Moreover, the exponential distribution used for the masses of each matrix element enables Gibbs sampling as described below and derived in detail previously [2, 4, 5, 6].

1.1 Algorithm Overview

Let $\mathcal{A}$ and $\mathcal{P}$ be the atomic domains for the A and P matrices. Each of the atomic domains is a set of 2-tuples (l, x) where $l \in \mathbb{N}$ is the position of the atom and $x \in \mathbb{R}_+$ is the mass of the atom (Fig 1). In practice, we restrict the length of the atomic domain so that $0 \leq l \leq 2^{64}$. Let $|\mathcal{A}|$ and $|\mathcal{P}|$ be the number of atoms in each domain. We also define $\mathcal{A}_{length}$ and $\mathcal{P}_{length}$ to be the size, or number of available positions, of each domain. The size is equal to the closest number to 2^{64} that the number of bins, or matrix elements, evenly divides. This allows each atom to be uniformly distributed and each matrix element to have an equal sized portion of the atomic domain mapping to it.

As shown in Algorithm 1, update steps are run in batches, alternating between the A and P matrix. An iteration of CoGAPS is defined as a batch of updates for each the A and P matrix. The algorithm runs in two phases, equilibration and sampling. Each of these phases is run for a pre-specified number of iterations, input by the user. The goal is to pick a number of iterations large enough for the sampling phase to obtain samples from the posterior distribution, $p(A, P|D, \sigma)$. These samples are averaged to get the final values for A and P as the Bayes minimum mean square error estimator.

Algorithm 1: CoGAPS

Input: $D, \sigma \in \mathbb{R}^{N \times M}$, $n, K \in \mathbb{N}$, $\alpha \in \mathbb{R}_+$
Output: $A \in \mathbb{R}_+^{N \times K}$, $P \in \mathbb{R}_+^{M \times K}$

`/* Initialize the A and P matrices along with their atomic domains */`
$A := 0_{N,K}, P := 0_{M,K}$
$\mathcal{A} := \varnothing, \mathcal{P} := \varnothing$

`/* Run the calibration phase of the algorithm */`
for $i = 1...n$ **do**
  $\tau = \min(1, 2i/n)$ `// Annealing Temperature`
  $n_A \leftarrow Poisson(|\mathcal{A}|), n_P \leftarrow Poisson(|\mathcal{P}|)$
  do update$(A, \mathcal{A}, \tau)$ n_A **times**
  do update$(P, \mathcal{P}, \tau)$ n_P **times**
end

`/* Run the sampling phase of the algorithm */`
$A_{final}, P_{final} := 0$
for $i = 1...n$ **do**
  $n_A \leftarrow Poisson(|\mathcal{A}|), n_P \leftarrow Poisson(|\mathcal{P}|)$
  do update$(A, \mathcal{A}, 1)$ n_A **times**
  do update$(P, \mathcal{P}, 1)$ n_P **times**
  A_{final} += A, P_{final} += P
end

`/* Return the estimate of the mean of the posterior distribution */`
return A_{final}/n, P_{final}/n

When updating the A and P matrices, CoGAPS preforms sampling steps to directly update the values of the atoms in $\mathcal{A}$ and $\mathcal{P}$. There are four types of updates that can happen in the atomic domain: 1) Birth - creates a new atom 2) Death - destroys an existing atom 3) Move - changes the location of an existing atom 4) Exchange - takes a portion of the mass from one atom and combines it with another. These four steps are selected as the updates for the atomic domain because they preserve the desired prior distributions of the matrix elements and enable Gibbs sampling. Each update step starts by randomly selecting one of these 4 changes. In any given update step, either one or two matrix elements will change. In the Bayesian NMF algorithm presented in [7], Gibbs sampling is done by sampling from the conditional posterior distribution of the matrix element, i.e. from $p(A_{i,j}|A_{\setminus(i,j)}, P, D, \sigma)$ or $p(P_{i,j}|P_{\setminus(i,j)}, A, D, \sigma)$. Due to the atomic domain in CoGAPS, sampling is instead done on the atoms themselves, i.e. sampling is done from $p(\Delta A_{i,j}|A, P, D, \sigma)$ or $p(\Delta P_{i,j}|P, A, D, \sigma)$. As described above, the sparsity of the matrix elements is determined by the Poission prior with parameter α. This distribution can be sampled from by altering the relative probability of a Birth or a Death step according to the sampling steps in Algorithm 2.

Algorithm 2: Update

`/* M is either A or P, M is the associated atomic domain */`

Input: $M, \mathcal{M}, \tau$

$u_1 \leftarrow Uniform(0,1), u_2 \leftarrow Uniform(0,1)$
if $u_1 < 1/2$ **then**
  $p := \dfrac{|\mathcal{M}|\mathcal{M}_{length}}{|\mathcal{M}|\mathcal{M}_{length} + \alpha|M|(\mathcal{M}_{length} - |\mathcal{M}|)}$
  if $u_2 < p$ **then**
    death$(M, \mathcal{M}, \tau)$
  else
    birth$(M, \mathcal{M}, \tau)$
  end
else
  if $u_2 < 1/2$ **then**
    move$(M, \mathcal{M}, \tau)$
  else
    exchange$(M, \mathcal{M}, \tau)$
  end
end

1.2 The Update Steps

As we describe above, there are four types of updates to the atomic domain in CoGAPS: birth, death, move, and exchange. The probability of an update step being any of these is described in Algorithms 1 and 2. In this section, we will describe how each step works. We break down each step into two parts: the proposal and the evaluation. This distinction becomes important for when we present the asynchronous updating scheme in the next section. The proposal phase determines which atoms are affected and the evaluation phase determines how large the effect is, and whether or not to accept certain changes. Note that we present the following math in terms of M and $\mathcal{M}$ for the matrix and atomic domain, since these steps are the same for either the A or the P matrix. We also reference quantities s and s_μ which are defined in the Appendix, with full derivations available in previous work [6].

1.2.1 Birth

In this type of update we are creating a new atom in a random location of the atomic domain.

Proposal

The proposal part of this step involves selecting a random, open location for the atom.

$l \leftarrow \mathcal{U}\{0, \mathcal{M}_{length}\}$ such that $(l, \cdot) \notin \mathcal{M}$

Let $l \mapsto r, q$, where r, q are the corresponding row and column in the matrix.

Evaluation

We want to sample a mass for this atom x. This corresponds to a change in the $M_{r,q}$ element, i.e. $\Delta M_{r,q} = x$ which is exponentially distributed. We want to sample x from $p(\Delta M_{r,q}|A, P, D, \sigma)$. Previously, we have shown that under the normal likelihood this distribution follows a truncated normal with the following parameters:

$$\mathcal{TN}_{(0,\cdot)}\left(\frac{2s\mu_{r,q}^{(M)} - \lambda}{2s_{r,q}^{(M)}}, \frac{1}{2\sqrt{s_{r,q}^{(M)}}}\right).$$

We sample x from this distribution and add the new atom, (l, x), to $\mathcal{M}$.

1.2.2 Death

In this type of update, we select an atom at random and remove it from the atomic domain. In practice, we take an additional step that helps the algorithm converge faster. Rather than just removing the atom outright, we give it a chance to re-sample a new mass and then keep that atom with some probability based on how the new mass effects the likelihood, $p(D|A, P, \sigma)$.

Proposal

In order to propose a death step, we only need to select an atom uniformly at random. Let (l, x_0) be the location and mass of this atom, and let $l \mapsto r, q$ where r, q are the corresponding row and column of the matrix.

Evaluation

When evaluating the death step, we need to first remove the atom from the atomic domain and then sample a new mass in the same location. Rather than actually removing the mass and updating the matrix, we can instead directly compute $p(\Delta M_{r,q}|x_0, A, P, D, \sigma)$ where x_0 is the original mass that has been removed. This distribution is a truncated normal with the following parameters:

$$\mathcal{TN}_{(0,\cdot)}\left(\frac{2s\mu_{r,q,x_0}^{(M)} - \lambda}{2s_{r,q}^{(M)}}, \frac{1}{2\sqrt{s_{r,q}^{(M)}}}\right)$$

We sample x from this distribution to get a new atom, (l, x). However, we only add it to $\mathcal{M}$ with some probability. Otherwise, the atom is removed from the atomic domain completely. The probability of keeping this new atom is $\exp\{\Delta\mathcal{L}(x, x_0)\}$, where $\Delta\mathcal{L}(x, x_0)$ is defined as the log-likelihood ratio:

$$\Delta\mathcal{L}(x, x_0) = \log(p(D|\Delta M_{r,q} = x - x_0, \sigma, A, P)) - \log(p(D|\Delta M_{r,q} = -x_0, \sigma, A, P)) = \frac{x(s\mu_{r,q,x_0}^{(M)} - xs_{r,q}^{(M)})}{2}$$

1.2.3 Move

In this type of update, we are moving an atom from one location to another. We restrict the move so that the atomic domain is not re-ordered, i.e. an atom can only move between it's left and right neighbors.

Proposal

We select an atom uniformly at random. Let (l_0, x) be the location and mass of this atom. Let l_{left} and l_{right} be the locations of the left and right neighbors of this atom. If the neighbor does not exist then $l_{left} = 0$ and $l_{right} = \mathcal{M}_{length}$. We then sample a new location l from the uniform distribution:

$l \sim \mathcal{U}\{l_{left}, l_{right}\}$

Let $l_0 \mapsto r_0, q_0$ and $l \mapsto r, q$ be the corresponding row and column each location maps to.

Evaluation

If $r_0 = r$ and $q_0 = q$:

Automatically accept the change as there is no effect on the matrix, simply update the atom's position.

If $r_0 \neq r$ or $q_0 \neq q$:

Accept the move with probability, $\exp\{\Delta\mathcal{L}(l_0, l)\}$, where $\Delta\mathcal{L}(l_0, l)$ is defined as the log-likelihood ratio:

$$\Delta\mathcal{L}(l, l_0) = \log(p(D|\Delta M_{r,q} = x, \Delta M_{r_0,q_0} = -x, \sigma, AP)) - \log(p(D|\sigma, A, P)) = \frac{x(s\mu^{(M)}_{r_0,r,q_0,q} - xs^{(M)}_{r_0,r,q_0,q})}{2}$$

1.2.4 Exchange

In this type of update, we are exchanging mass between two atoms. First an atom is selected at random. Then we use a Gibbs sampling approach to determine how much mass to exchange between this atom and its right neighbor.

Proposal

We select an atom uniformly at random. Let (l_1, x_1) be this atom and let (l_2, x_2) be its right neighbor. If the right neighbor does not exist (i.e. l_1 is the right-most atom) then set the right neighbor equal to the left-most atom. Let $l_1 \mapsto r_1, q_1$ and $l_2 \mapsto r_2, q_2$ be the corresponding row and column each location maps to.

Evaluation

If $r_1 \neq r_2$ or $q_1 \neq q_2$:

We want to sample the mass increase to the first atom, let this be x (i.e. $\Delta x_1 = x$). Since we are exchanging mass, we must preserve the total mass between both atoms. To accomplish this we must have $x \in (-x_1, x_2)$. At the end of this step we will set the new masses to be $x_1 \to x_1 + x$ and $x_2 \to x_2 - x$. We want to sample x from $p(\Delta M_{r_1,q_1} = x, \Delta M_{r_2,q_2} = -x|D, \sigma, A, P)$. This distribution is a truncated normal with the following parameters:

$$\mathcal{TN}_{(-x_1,x_2)}\left(\frac{s\mu^{(M)}_{r_1,r_2,q_1,q_2}}{s^{(M)}_{r_1,r_2,q_1,q_2}}, \frac{1}{2\sqrt{s^{(M)}_{r_1,r_2,q_1,q_2}}}\right)$$

If $r_1 = r_2$ and $q_1 = q_2$:

In this case, there is no change to the matrix elements or the structure of the atomic domain so it is safe to ignore this update.

1.3 Annealing Temperature

When the algorithm is in the equilibration phase, we use a strategy known as simulated annealing. This allows the initial samples to be dominated by the prior distributions rather than the likelihood. Let τ be the temperature, then in all cases we have $s \leftarrow \tau s$ and $s\mu \leftarrow \tau s\mu$. Eventually we reach $\tau = 1$, in which case s and $s\mu$ are calculated exactly as described above.

2 Asynchronous Update Steps

The previous section outlined the standard CoGAPS algorithm and showed how the update steps of the algorithm work in detail. It is important to note that, in the standard algorithm, these update steps must be run sequentially. The proposal phase of each update step requires knowledge about the current state of the atomic domain. If the updates were run in parallel, it would not be possible to know the current state of the domain and the algorithm would be fundamentally different. The evaluation phase, however, is not so restricted. This phase is just a large matrix calculation. If two update steps involve unrelated segments of the matrices, then there is no reason the evaluation phase cannot be run in parallel. Conveniently, the proposal phase is relatively inexpensive to compute when compared to the evaluation phase. This concept is the foundation of the asynchronous version of CoGAPS. The proposal phase of the update steps is run sequentially in order to build up a queue of "independent" proposals which are then evaluated in parallel.

The details of the asynchronous algorithm come down to defining independent proposals. We want two proposals to be independent only if they can be evaluated in any order without changing the state of the algorithm (i.e. the atomic domain and matrix values). If we are able to do that, then the algorithm is straightforward: propose as many updates as possible until there is some dependency, then evaluate all the independent proposals in parallel. If our definition is correct, this will result in the same state of the algorithm as if we processed everything sequentially. Since it is difficult to define what makes proposals independent, we will instead make an exhaustive list of the conflicts that could prevent us from adding a new proposal to the queue. The next three sections explore these conflicts.

2.1 Conflict in Matrix Calculations

The first type of conflict arises in the calculations of s and $s\mu$. Both of the quantities depend on the values in the A and P matrices. For the sake of example, say that we are proposing a change to $A_{r,q}$. In this case, we will calculate $s^{(A)}_{r,q}$ and $s\mu^{(A)}_{r,q}$ which both depend of the r^{th} row of AP^T. If we accept a change to $A_{r,q}$ we will also be changing the entire r^{th} row of AP^T and this is where the conflict happens. If there are two proposals in the same row of A, then the outcome of one will effect the calculations of s and $s\mu$ for the other. Therefore, any two proposals that effect the same rows are not independent. This observation has been made in other matrix factorization algorithms and similiar schemes were designed in order to run the factorization parallel [7, 8, 9, 10]. To check for this conflict, we keep a table of all the rows that are currently used by proposals in the queue. If any new proposal has an overlap, we stop the queue and move on to the evaluation phase. Surprisingly, this conflict accounts for nearly all the possible conflicts that could arise, however there are some corner cases described in the next section.

2.2 Atomic Domain Ordering and Finding Neighbors

Checking for a matrix conflict will take care of most of the conflicts that can occur between proposals, however there is one type of update that can be problematic without causing any matrix conflicts. When doing a move step, we select an atom at random and move it somewhere between its neighbors. That means we must be able to calculate the positions of both neighbors. Most of the time, neighboring atoms will map to matrix elements in the same row, but this is not guaranteed to be the case - especially since the atomic domain can be very sparse. Therefore, it is possible that the starting location and proposed new location for a moved atom could map to matrix elements in a different row than the neighboring atoms. In this case, it would be possible for the neighbors to be involved in a previously queued update step but not have any conflicts in the matrix calculations. To account for this, during a move step we must explicitly check that the neighboring atoms are not involved in any previously queued updates.

There is additional type of conflict that can arise directly in the atomic domain during the birth step. Any queued death step can open up more locations for future atoms to be created in. The probability of a location chosen at random being already occupied is astronomically small (in practice the atomic domain has 2^{64} possible locations). The chance that not only was this location occupied, but the atom also was removed in the last handful of updates is so small that we ignore this conflict. New atoms are created among all the available locations, not counting ones that could be made available from the current queue.

2.3 Birth/Death Decision

Direct conflicts between update steps are not the only way that the queue can be stopped. It is possible to run into an issue before we even decide what type of update to queue next. When deciding between a birth and death step, we need to know the current size of the atomic domain. This allows the algorithm to enforce a sparsity constraint by tightly controlling the total number of atoms. Every time a proposed update is queued, it becomes less clear what the total number of atoms will be. At update t if we know there n atoms, and we propose a death step, then at $t+1$ there are between $n-1$ and n atoms. The more updates that are queued up, the larger this range becomes.

Fortunately, the function that determines the probability of death vs birth is a monotonic function of the total number of atoms. Let $P_{death}(n)$ be the death probability when there are n atoms and let n_t be the total number of atoms at time t. If we have evaluated all updates up to time t then we know n_t exactly. If we propose k more updates however, we do not know what n_{t+k} is. Based on the types of the proposed updates we can calculate a range for this value. Let N_{deaths} and N_{births} be the number of deaths and births proposed in the current queue. Then we know that $n_t - N_{deaths} + N_{births} \leq n_{t+k} \leq n_t + N_{births}$. We therefore know that $P_{death}(n_t - N_{deaths} + N_{births}) \leq P_{death}(n_{t+k}) \leq P_{death}(n_t + N_{births})$.

When making the decision between a birth step and a death step at time t, we draw $u \sim \mathcal{U}(0,1)$. If $u < P_{death}(n_t)$ we propose a death step, otherwise a birth step. For making this decision k proposals into the future, we have three possible outcomes instead of two. If $u < P_{death}(n_t - N_{deaths} + N_{births})$ then we know we have a death step. If $u > P_{death}(n_t + N_{births})$ then we know we have a birth step. If u is between these two values then we cannot make a decision and the result is indeterminate. We must first evaluate the current queue in order to get the exact value for n_{t+k}. So, in this case the queue stops being populated and we move on to the evaluation phase.

3 Sparse Data Optimization

The previous sections have discussed strategies improving the run time of CoGAPS, but have not addressed the amount of memory used by the algorithm. Memory overhead is one of the most significant problems facing the analysis of large single-cell datasets. Fortunately, these datasets tend to be extremely sparse and so there is an opportunity for reducing the memory overhead of the algorithm by storing the data in sparse data structures. Computing the values of s and $s\mu$ on sparse data structures however is not straightforward. It is necessary to derive a sparse-friendly version of the s and $s\mu$ calculations to avoid increasing run time in solving for memory usage.

3.1 Sparse Matrix Calculations

In this case we will derive the equations just for the A matrix, since the derivation for the P matrix is identical except for the labels. We first define the non-zero indices of a given row of the data as γ and make an assumption about the standard deviation, σ:

$$\text{Let } \gamma(i) = \{j : D_{ij} > 0\} \text{ and } \sigma_{ij} = \begin{cases} \sigma_0 D_{ij} & \text{if } D_{ij} > 0 \\ \sigma_0 & \text{if } D_{ij} = 0 \end{cases}$$

In the Appendix we show the full derivation of the following identities:

$$s^{(A)}_{r,q} = \sum_j \frac{P^2_{jq}}{2\sigma^2_{rj}} = \frac{1}{\sigma^2_0}\sum_j P^2_{jq} + \frac{1}{\sigma^2_0}\sum_{j\in\gamma(r)}\left(\frac{P^2_{jq}}{D^2_{rj}} - P^2_{jq}\right)$$

$$s\mu^{(A)}_{r,q} = \sum_j \frac{P_{jq}(D_{rj} - \sum_k A_{rk}P_{jk})}{2\sigma^2_{rj}} = \frac{-\sum_k A_{rk}\sum_l P_{lq}P_{lk}}{\sigma^2_0} + \frac{1}{\sigma^2_0}\sum_{j\in\gamma(r)}\frac{P_{jq}}{D_{rj}} + \left(P_{jq} - \frac{P_{jq}}{D^2_{ij}}\right)\sum_k A_{rk}P_{jk}$$

$$s\mu^{(A)}_{r,q,x} = \sum_j \frac{P_{jq}(D_{rj} - \sum_k A_{rk}P_{jk} - xP_{jq})}{2\sigma^2_{rj}}$$

$$= \frac{-x\sum_j P_{jq}^2 - \sum_k A_{rk}\sum_j P_{jq}P_{jk}}{\sigma_0^2} + \frac{1}{\sigma_0^2}\sum_{j\in\gamma(r)}\frac{P_{jq}}{D_{rj}} + \left(P_{jq} - \frac{P_{jq}}{D_{ij}^2}\right)\left(\sum_k A_{rk}P_{jk} + xP_{jq}\right)$$

(This is for $r = r_1 = r_2$, the $r_1 \neq r_2$ case requires no additional derivation)

$$s^{(A)}_{r_1,r_2,q_1,q_2} = \sum_j \frac{(P_{jq_1} - P_{jq_2})^2}{2\sigma_{rj}^2} = s^{(A)}_{r,q_1} + s^{(A)}_{r,q_2} - \frac{2\sum_j P_{jq_1}P_{jq_2}}{\sigma_0^2} + \frac{2}{\sigma_0^2}\sum_{j\in\gamma(r)}\left(P_{jq_1}P_{jq_2} - \frac{P_{jq_1}P_{jq_2}}{D_{rj}^2}\right)$$

Note that we can longer use the AP^T matrix in the calculations since it is not feasible to cache it when working with large, sparse datasets. Even when D is very sparse, the product of A and P will not be. In fact, AP^T is nearly all non-zero is most real world cases. Storing this large of a matrix defeats the entire purpose of the optimization, so we must compute AP^T on the fly.

3.2 Runtime Complexity

Each of these calculations follow a similar pattern: there is some initial term and then a sum over the non-zero indices of the data. Without the optimization, we would have to sum over all the indices of the data. Therefore, as long as the initial term doesn't take too long to compute, the full computation should be faster than the standard computation by a factor of the data sparsity. In this section, we will explore the runtime complexity and computational strategies for the initial terms.

$$s^{(A)}_{r,q} \to \frac{1}{\sigma_0^2}\sum_j P_{jq}^2$$

This term only depends on the P matrix which is constant while A is being updated, so it can be pre-computed ahead of a batch update step for A. To compute this term for all values of q requires $O(MK)$ time where M is the number of samples and K is the number of patterns. A batch of updates can be done to A is $O(NMK)$ time where N is the number of features. Thus creating a lookup table for this initial term has a negligible cost when compared to the full batch of updates. Once we have a lookup table, then computing the initial term takes $O(1)$.

$$s\mu^{(A)}_{r,q} \to \frac{-\sum_k A_{rk}\sum_j P_{jq}P_{jk}}{\sigma_0^2}$$

The $\sum_j P_{jq}P_{jk}$ term only depends on P so we can also make a lookup table for this value. Creating this lookup table takes $O(MK^2)$ time, whereas the full batch of updates takes $O(MNK)$. In practice, $K << N$ so this is an acceptable overhead to incur. Once we have a lookup table, the initial term takes $O(K)$ to compute, which makes the update step go from $O(MK)$ to $O(MK+K)$ which again is acceptable since $K << M$.

$$s\mu^{(A)}_{r,q,x} \to \frac{-x\sum_j P_{jq}^2 - \sum_k A_{rk}\sum_j P_{jq}P_{jk}}{\sigma_0^2}$$

We have already created a lookup table for both $\sum_j P_{jq}^2$ and $\sum_j P_{jq}P_{jk}$ so there is no additional cost for this step, and the remaining computation takes the same amount of time as the previous quantity, $s\mu^{(A)}_{r,q}$, so no additional work needs to be done to show this incurs an acceptable overhead.

$$s^{(A)}_{r_1,r_2,q_1,q_2} \to s^{(A)}_{r,q_1} + s^{(A)}_{r,q_2} - \frac{2\sum_j P_{jq_1}P_{jq_2}}{\sigma_0^2}$$

While it may seem that $s^{(A)}_{r,q_1} + s^{(A)}_{r,q_2}$ are part of the initial term, both of these quantities can be computed alongside the main term, so no additional run time cost is incurred. The $\sum_j P_{jq_1}P_{jq_2}$ term has already been addressed above, so there is no additional work needed.

4 Appendix

4.1 Single Matrix Element Calculations

These calculations are relevant when a single matrix element is being changed (such as in birth or death). Let r, q be the row and column of the changed element and let $^{(A)}$ or $^{(P)}$ denote which matrix is being changed.

$$s_{r,q}^{(A)} = \sum_j \frac{P_{jq}^2}{2\sigma_{rj}^2} \qquad s_{r,q}^{(P)} = \sum_i \frac{A_{iq}^2}{2\sigma_{ir}^2}$$

$$s\mu_{r,q}^{(A)} = \sum_j \frac{P_{jq}(D_{rj} - (AP^T)_{rj})}{2\sigma_{rj}^2} \qquad s\mu_{r,q}^{(P)} = \sum_i \frac{A_{iq}(D_{ir} - (AP^T)_{ir})}{2\sigma_{ir}^2}$$

In this case, we want to make the same $s\mu$ calculation, but with the assumption that a change x has already been made to the (r, q) matrix element.

$$s\mu_{r,q,x}^{(A)} = \sum_j \frac{P_{jq}(D_{rj} - (AP^T)_{rj} - xP_{jq})}{2\sigma_{rj}^2} \qquad s\mu_{r,q,x}^{(P)} = \sum_i \frac{A_{iq}(D_{ir} - (AP^T)_{ir} - xA_{iq})}{2\sigma_{ir}^2}$$

4.2 Multiple Matrix Element Calculations

These calculations are relevant when two matrix elements are being changed at the same time (such as in move or exchange). Let r_1, q_1 be the row and column of the first element and r_2, q_2 be the second element.

$$s\mu_{r_1,r_2,q_1,q_2}^{(A)} = s\mu_{r_1,q_1}^{(A)} - s\mu_{r_2,q_2}^{(A)} \qquad s\mu_{r_1,r_2,q_1,q_2}^{(P)} = s\mu_{r_1,q_1}^{(P)} - s\mu_{r_2,q_2}^{(P)}$$

If $r_1 = r_2$:

$$s_{r_1,r_2,q_1,q_2}^{(A)} = \sum_j \frac{(P_{jq_1} - P_{jq_2})^2}{2\sigma_{r_1 j}^2} \qquad s_{r_1,r_2,q_1,q_2}^{(P)} = \sum_i \frac{(A_{iq_1} - A_{iq_2})^2}{2\sigma_{ir_1}^2}$$

If $r_1 \neq r_2$:

$$s_{r_1,r_2,q_1,q_2}^{(A)} = s_{r_1,q_1}^{(A)} + s_{r_2,q_2}^{(A)} \qquad s_{r_1,r_2,q_1,q_2}^{(P)} = s_{r_1,q_1}^{(P)} + s_{r_2,q_2}^{(P)}$$

4.3 Sparse Matrix Calculations

Let $\gamma(i) = \{j : D_{ij} > 0\}$ and $\sigma_{ij} = \begin{cases} \sigma_0 D_{ij} & \text{if } D_{ij} > 0 \\ \sigma_0 & \text{if } D_{ij} = 0 \end{cases}$

4.3.1 $s_{r,q}$

$$\begin{aligned}
s_{r,q}^{(A)} &= \sum_j \frac{P_{jq}^2}{2\sigma_{rj}^2} \\
&= \frac{1}{2\sigma_0^2} \sum_{j\in\gamma(r)} \frac{P_{jq}^2}{D_{rj}^2} + \frac{1}{2\sigma_0^2} \sum_{j\notin\gamma(r)} P_{jq}^2 \\
&= \frac{1}{2\sigma_0^2} \sum_{j\in\gamma(r)} \frac{P_{jq}^2}{D_{rj}^2} + \frac{1}{2\sigma_0^2} \left(\sum_j P_{jq}^2 - \sum_{j\in\gamma(r)} P_{jq}^2 \right) \\
&= \frac{1}{2\sigma_0^2} \sum_{j\in\gamma(r)} \frac{P_{jq}^2}{D_{rj}^2} - \frac{1}{2\sigma_0^2} \sum_{j\in\gamma(r)} P_{jq}^2 + \frac{1}{2\sigma_0^2} \sum_j P_{jq}^2 \\
&= \frac{1}{2\sigma_0^2} \sum_j P_{jq}^2 + \frac{1}{2\sigma_0^2} \sum_{j\in\gamma(r)} \left(\frac{P_{jq}^2}{D_{rj}^2} - P_{jq}^2 \right)
\end{aligned}$$

4.3.2 $s\mu_{r,q}$

$$s\mu_{r,q}^{(A)} = \sum_j \frac{P_{jq}(D_{rj} - \sum_k A_{rk}P_{jk})}{2\sigma_{rj}^2}$$
$$= \sum_{j\in\gamma(r)} \frac{P_{jq}(D_{rj} - \sum_k A_{rk}P_{jk})}{2\sigma_0^2 D_{rj}^2} - \sum_{j\notin\gamma(r)} \frac{P_{jq}}{2\sigma_0^2} \sum_k A_{rk}P_{jk}$$
$$= \sum_{j\in\gamma(r)} \frac{P_{jq}(D_{rj} - \sum_k A_{rk}P_{jk})}{2\sigma_0^2 D_{rj}^2} - \sum_k \frac{A_{rk}}{2\sigma_0^2} \sum_{j\notin\gamma(r)} P_{jq}P_{jk}$$
$$= \sum_{j\in\gamma(r)} \frac{P_{jq}(D_{rj} - \sum_k A_{rk}P_{jk})}{2\sigma_0^2 D_{rj}^2} - \sum_k \frac{A_{rk}}{2\sigma_0^2} \left(\sum_j P_{jq}P_{jk} - \sum_{j\in\gamma(r)} P_{jq}P_{jk} \right)$$
$$= \sum_{j\in\gamma(r)} \frac{P_{jq}(D_{rj} - \sum_k A_{rk}P_{jk})}{2\sigma_0^2 D_{rj}^2} + \sum_k \frac{A_{rk}}{2\sigma_0^2} \sum_{j\in\gamma(r)} P_{jq}P_{jk} - \sum_k \frac{A_{rk}}{2\sigma_0^2} \sum_j P_{jq}P_{jk}$$
$$= \frac{-\sum_k A_{rk} \sum_j P_{jq}P_{jk}}{2\sigma_0^2} + \sum_{j\in\gamma(r)} \frac{P_{jq}(D_{rj} - \sum_k A_{rk}P_{jk})}{2\sigma_0^2 D_{rj}^2} + \sum_{j\in\gamma(r)} \frac{P_{jq}}{2\sigma_0^2} \sum_k A_{rk}P_{jk}$$
$$= \frac{-\sum_k A_{rk} \sum_j P_{jq}P_{jk}}{2\sigma_0^2} + \frac{1}{2\sigma_0^2} \sum_{j\in\gamma(r)} \frac{P_{jq}}{D_{rj}} + \left(P_{jq} - \frac{P_{jq}}{D_{ij}^2} \right) \sum_k A_{rk}P_{jk}$$

4.3.3 $s\mu_{r,q,x}$

$$s\mu_{r,q,x}^{(A)} = \sum_j \frac{P_{jq}(D_{rj} - \sum_k A_{rk}P_{jk} - xP_{jq})}{2\sigma_{rj}^2}$$
$$= \sum_{j\in\gamma(r)} \frac{P_{jq}(D_{rj} - \sum_k A_{rk}P_{jk} - xP_{jq})}{2\sigma_0^2 D_{rj}^2} - \sum_{j\notin\gamma(r)} \frac{P_{jq}}{2\sigma_0^2} \left(\sum_k A_{rk}P_{jk} + xP_{jq} \right)$$
$$= \sum_{j\in\gamma(r)} \frac{P_{jq}(D_{rj} - \sum_k A_{rk}P_{jk} - xP_{jq})}{2\sigma_0^2 D_{rj}^2} - \frac{x}{2\sigma_0^2} \sum_{j\notin\gamma(r)} P_{jq}^2 - \sum_k \frac{A_{rk}}{2\sigma_0^2} \sum_{j\notin\gamma(r)} P_{jq}P_{jk}$$
$$= \frac{-x\sum_j P_{jq}^2 - \sum_k A_{rk} \sum_j P_{jq}P_{jk}}{2\sigma_0^2} + \frac{1}{2\sigma_0^2} \sum_{j\in\gamma(r)} \frac{P_{jq}}{D_{rj}} - \frac{P_{jq}\left(\sum_k A_{rk}P_{jq} + xP_{jq} \right)}{D_{rj}^2} + P_{jq}\left(\sum_k A_{rk}P_{jk} + xP_{jq} \right)$$
$$= \frac{-x\sum_j P_{jq}^2 - \sum_k A_{rk} \sum_j P_{jq}P_{jk}}{2\sigma_0^2} + \frac{1}{2\sigma_0^2} \sum_{j\in\gamma(r)} \frac{P_{jq}}{D_{rj}} + \left(P_{jq} - \frac{P_{jq}}{D_{rj}^2} \right) \left(\sum_k A_{rk}P_{jq} + xP_{jq} \right)$$

4.3.4 $s_{r_1,r_2,q_1,,q_2}$

In this case, $r = r_1 = r_2$

$$s_{r_1,r_2,q_1,q_2}^{(A)} = \sum_j \frac{(P_{jq_1} - P_{jq_2})^2}{2\sigma_{rj}^2}$$
$$= \sum_j \frac{P_{jq_1}^2}{2\sigma_{rj}^2} + \sum_j \frac{P_{jq_2}^2}{2\sigma_{rj}^2} - \sum_j \frac{P_{jq_1}P_{jq_2}}{\sigma_{rj}^2}$$
$$= s_{r,q_1}^{(A)} + s_{r,q_2}^{(A)} - \left(\sum_{j\in\gamma(r)} \frac{P_{jq_1}P_{jq_2}}{\sigma_{rj}^2} + \sum_{j\notin\gamma(r)} \frac{P_{jq_1}P_{jq_2}}{\sigma_{rj}^2} \right)$$
$$= s_{r,q_1}^{(A)} + s_{r,q_2}^{(A)} - \left(\sum_{j\in\gamma(r)} \frac{P_{jq_1}P_{jq_2}}{\sigma_0^2 D_{rj}^2} + \sum_{j\notin\gamma(r)} \frac{P_{jq_1}P_{jq_2}}{\sigma_0^2} \right)$$
$$= s_{r,q_1}^{(A)} + s_{r,q_2}^{(A)} - \sum_{j\in\gamma(r)} \frac{P_{jq_1}P_{jq_2}}{\sigma_0^2 D_{rj}^2} - \left(\sum_j \frac{P_{jq_1}P_{jq_2}}{\sigma_0^2} - \sum_{j\in\gamma(r)} \frac{P_{jq_1}P_{jq_2}}{\sigma_0^2} \right)$$
$$= s_{r,q_1}^{(A)} + s_{r,q_2}^{(A)} - \sum_j \frac{P_{jq_1}P_{jq_2}}{\sigma_0^2} - \sum_{j\in\gamma(r)} \frac{P_{jq_1}P_{jq_2}}{\sigma_0^2 D_{rj}^2} - \frac{P_{jq_1}P_{jq_2}}{\sigma_0^2}$$

4.4 Sparse χ^2 Calculation

The following derivation shows a more sparse-friendly way of calculating χ^2. This isn't relevant for the updating portion of the algorithm, but the performance is somewhat relevant since it is calculated periodically throughout the run of the algorithm.

$$\chi^2 = \sum_i \chi_i^2$$

$$\chi_i^2 = \sum_j \frac{(D_{ij} - \sum_l A_{il}P_{jl})^2}{\sigma_{ij}^2}$$

$$= \sum_{j\in\gamma(i)} \frac{(D_{ij} - \sum_l A_{il}P_{jl})^2}{\sigma_0^2 D_{ij}^2} + \frac{1}{\sigma_0^2}\sum_{j\notin\gamma(i)}(\sum_l A_{il}P_{jl})^2$$

$$= \sum_{j\in\gamma(i)} \frac{(D_{ij} - \sum_l A_{il}P_{jl})^2}{\sigma_0 D_{ij}^2} + \frac{1}{\sigma_0^2}\sum_j(\sum_l A_{il}P_{jl})^2 - \frac{1}{\sigma_0^2}\sum_{j\in\gamma(i)}(\sum_l A_{il}P_{jl})^2$$

$$= \sum_{j\in\gamma(i)} \frac{(D_{ij} - \sum_l A_{il}P_{jl})^2}{\sigma_0^2 D_{ij}^2} - \frac{(\sum_l A_{il}P_{jl})^2}{\sigma_0^2} + \frac{1}{\sigma_0^2}\sum_j(\sum_l A_{il}P_{jl})^2$$

$$= \frac{1}{\sigma_0^2}\sum_{j\in\gamma(i)} \frac{(D_{ij} - \sum_l A_{il}P_{jl})^2}{D_{ij}^2} - (\sum_l A_{il}P_{jl})^2 + \frac{1}{\sigma_0^2}\sum_j(\sum_l A_{il}P_{jl})^2$$

$$= \frac{1}{\sigma_0^2}\sum_{j\in\gamma(i)} \frac{D_{ij}^2 + (\sum_l A_{il}P_{jl})^2 - 2D_{ij}\sum_l A_{il}P_{jl}}{D_{ij}^2} - (\sum_l A_{il}P_{jl})^2 + \frac{1}{\sigma_0^2}\sum_j(\sum_l A_{il}P_{jl})^2$$

$$= \frac{1}{\sigma_0^2}\sum_{j\in\gamma(i)} \frac{D_{ij}^2 + (\sum_l A_{il}P_{jl})^2 - 2D_{ij}\sum_l A_{il}P_{jl} - D_{ij}^2(\sum_l A_{il}P_{jl})^2}{D_{ij}^2} + \frac{1}{\sigma_0^2}\sum_j(\sum_l A_{il}P_{jl})^2$$

$$= \frac{1}{\sigma_0^2}\sum_{j\in\gamma(i)} \frac{D_{ij}^2 + (\sum_l A_{il}P_{jl})(\sum_l A_{il}P_{jl} - 2D_{ij} - D_{ij}^2\sum_l A_{il}P_{jl})}{D_{ij}^2} + \frac{1}{\sigma_0^2}\sum_j(\sum_l A_{il}P_{jl})^2$$

$$= \frac{1}{\sigma_0^2}\sum_{j\in\gamma(i)} 1 + \frac{(\sum_l A_{il}P_{jl})(\sum_l A_{il}P_{jl} - 2D_{ij} - D_{ij}^2\sum_l A_{il}P_{jl})}{D_{ij}^2} + \frac{1}{\sigma_0^2}\sum_j(\sum_l A_{il}P_{jl})^2$$

References

[1] Sibusiso Sibisi, John Skilling. (1997). *Prior Distributions on Measure Space*. J. R. Stat. Soc. B 59, 217-235.

[2] Elana J. Fertig, Jie Ding, Alexander V. Favorov, Giovanni Parmigiani, Michael F. Ochs. (2010). *CoGAPS: an R/C++ package to identify patterns and biological process activity in transcriptomic data*. Bioinformatics, 26(21): 2792-2793.

[3] Bo Li et al. (2018). Census of Immune Cells. Broad Inst. Mass. Inst. Technol. Howard Hughes Med. Inst. Retrieved from https://data.humancellatlas.org/explore/projects/cc95ff89-2e68-4a08-a234-480eca21ce79

[4] Andrew V. Kossenkov, Aidan J. Peterson, Michael F. Ochs. (2007). *Determining Transcription Factor Activity from Microarray Data using Bayesian Markov Chain Monte Carlo Sampling*. Stud. Health Technol. Inform. 129: 1250-1254.

[5] Elana J. Fertig, Michael F. Ochs. (2012). *Matrix factorization for transcriptional regulatory network inference*. IEEE Symp. Comput. Intell. Bioinforma Comput. Biol. Proc. 387-396.

[6] Genevieve L. Stein-O'Brien, Brian S. Clark, Thomas Sherman, Cristina Zibetti, Qiwen Hu, Rachel Sealfon, Sheng Liu, Jiang Qian, Carlo Colantuoni, Seth Blackshaw, Loyal A. Goff, Elana J. Fertig. (2019). *Decomposing Cell Identity for Transfer Learning across Cellular Measurements, Platforms, Tissues, and Species*. Cell Systems. 8(5): 395-411.

[7] Mikkel N. Schmidt, Ole Winther, Lars Kai Hansen. (2009). *Bayesian Non-negative Matrix Factorization*. Independent Component Analysis and Signal Separation. 540-547.

[8] Guangyong Chen, Fengyuan Zhu, Pheng A. Heng. (2018). *Large-Scale Bayesian Probabilistic Matrix Factorization with Memo-Free Distributed Variational Inference*. ACM Trans. Knowl. Discov. Data. 12, 3, Article 31, 24 pages.

[9] Sungjin Ahn, Anoop Korattikara, Nathan Liu, Suju Rajan, Max Welling. (2015). *Large Scale Distributed Bayesian Matrix Factorization using Stochastic Gradient MCMC*. Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. Pages 9-18.

[10] Fanglin Li, Bin Wu, Liutong Xu, Chuan Shi, Jing Shi. (2014). *A Fast Distributed Stochastic Gradient Descent Algorithm for Matrix Factorization.* JMLR: Workshop and Conference Proceedings 36: 77-87.

[11] Stein-O'Brien,G.L. et al. (2017). *PatternMarkers GWCoGAPS for novel data-driven biomarkers via whole transcriptome NMF.* Bioinforma. Oxf. Engl., 33, 1892–1894.