Computational proposal for sparse matrix

avrahami.ben

March 2020

1 Introduction

Given a population size N, Client requires two data structures:

- a true sparse matrix (we will call S_0)
- a linear combination of S_0's (we will call S_1)

2 S_0 desired operations

1. matrix: S_0 should be able to store and access positive real elements mapped by $(0..N-1)^2$
2. space efficient: Even though the previous requirement should require $O(N^2)$ space, we expect that S_0 will only store $O(N)$ non-zero elements at any time, should it should be optimized to use only $O(N)$ space.
3. set: S_0 will be mutable, and we should be able to set a cell in it with a positive real value.

3 S_1 desired operations

1. linear combinations: S_1 should be constructable using a set of S_0's and positive real coefficients.
2. random access: S_1 should be able to access a linear combination of its S_0 elements at any coordinate.
3. space efficient: As with S_0 we expect S_1 to only store $O(N)$ non-zero elements, and should only require $O(N)$ space.
4. total sum: we will need access to the sums of all elements of S_1 (this value is then divided by N, but the client can to that).
5. scalar multiplication: S_1 must support multiplication by a real positive scalar, effectively multiplying all its element by the scalar.

6. probability of any: given a real $v_{1\times N}$ s.t. $\forall_j v[j] \in [0,1]$, we want to calculate
   $1 - \prod_{j\in(0..N-1)} (1 - S_1[i,j] * v[j])$
   for all $i < N$.

7. multiply sub-matrix row by scalar: given a row r, an S_0 component matrix, and a real scalar s, we want to multiply all the values of row r in S_0 by s, and reflect the results in S_1. It is unclear whether this operation should be reversible, and whether s will ever differ from 0.

8. multiply sub-matrix column by scalar: given a column c, an S_0 component matrix, and a real scalar s, we want to multiply all the values of column c in S_0 by s, and reflect the results in S_1. It is unclear whether this operation should be reversible, and whether s will ever differ from 0.

working assumption We will assume there exists $T \ll N$ s.t. in every row in S_1 and S_0, there are no more than T non-zero elements.

4 Implementation Proposal: nested dictionaries

4.1 S_0

S_0 will be implemented as an array of sorted maps, mapping each row number to a row that stores all its cells in a sorted map. Each matrix will store its total sum. Each matrix will also store, per column, a set of all its non-zero indices (the average size of these sets will be less than T).

example- the matrix:

$$\begin{bmatrix} 1 & 0 & 0 & 2 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 2 & 0 \end{bmatrix}$$

will be represented as:

$$data = \left[\begin{pmatrix} 0 \mapsto 1 \\ 3 \mapsto 2 \end{pmatrix}, (2 \mapsto 1), \begin{pmatrix} 0 \mapsto 1 \\ 1 \mapsto 3 \end{pmatrix}, (4 \mapsto 1), \begin{pmatrix} 0 \mapsto 1 \\ 1 \mapsto 1 \\ 3 \mapsto 2 \end{pmatrix} \right]$$

$$columns = \left[\begin{Bmatrix} 0 \\ 2 \\ 4 \end{Bmatrix}, \begin{Bmatrix} 2 \\ 4 \end{Bmatrix}, \{1\}, \begin{Bmatrix} 0 \\ 4 \end{Bmatrix}, \{3\}, \right]$$

$$total = 13$$

sorted map implementation the most basic implementation for sorted maps is a binary tree, but more optimal implementations can be swapped-in down the line.

space S_0 will require at most $O(T * N)$ space

setting setting a value is trivial and will require $O(log(T))$ time

batch setting setting can be optimized for many values in a single row, setting $M > \frac{T}{log(T)-1}$ new values to a single row will only require $O(M+T)$ time (instead of $O(log(T) * M)$)

multiply row by scalar in-place S_0 matrices will support multiplying the values of a row in place by a scalar. Trivial (irreversable) implementation $O(T)$.

multiply column by scalar in-place S_0 matrices will support multiplying the values of a column in place by a scalar. Trivial (irreversible) implementation $O(log(T) * T)$.

4.2 S_1

S_1 will store all its component S_0 matrices, along with the coefficient for each. Upon construction, it will also create and store an additional S_0 matrix, filled with the actual values of S_1 for quick access.

invariant we will assume that all modifications to the underlying S_0 components will occur through S_1.

construction for an S_1 with K component matrices, construction requires $O(K * T * N)$ time.

space S_1 will only require as much space as its component matrices, as well as an additional $O(T * N)$ space for the cached matrix.

total sum trivially accessible from the cached matrix.

scalar multiplication this will require to simply multiply all the coefficients of the component matrices and to reconstruct the cached matrix.

probability of any: see later section.

multiply sub-matrix row by scalar: this will be done by multiplying the component row in-place and recalculating the cached matrix only along that row (total time $O(T)$).

multiply sub-matrix column by scalar: this will be done by multiplying the component column in-place and recalculating the cached matrix only along that column (total time $O(log(T) * N)$).

5 Implementing probability of any

On its face, the "probability of any" problem

$$POA(S_1, v, r) = 1 - \prod_{j \in (0..N-1)} (1 - S_1[r, j] * v[j]) \tag{1}$$

is one that is quite difficult to solve both computationally and mathematically.

assumption on v We will assume that v is sparse. Assume there exists $K \ll N$ s.t. there are no more than K non-zero elements in v.

mathematical solutions Both Tzvi and Tal suggested solutions that operate on the entire S_1 matrix in batch. These would be ideal in cases of a full matrix (like in numpy), but on sparese matrices, batch operations are slower, and require more space (simply re-calculating all the additional data will require an extra $O(N * T)$ space and time whenever the matrix changes.

computational solution I propose a computational solution that utilizes the assumption that both S_1 and V are sparse. Runs in $O(T+K)$ time per row.

Algorithm 1: computational POA algorithm

Result: $POA(S_1, V, r)$

$I \leftarrow$All non-zero indices of $S_1[r,:]$, queued and sorted;
$J \leftarrow$All non-zero indices of V, queued and sorted;
$result \leftarrow 1$;
while I *is not empty and* J *is not empty***:**
    $i \leftarrow I[0]$;
    $j \leftarrow J[0]$;
    if $i < j$**:**
        remove $I[0]$ from I
    elif $j < i$**:**
        remove $J[0]$ from J
    else:
        # $i = j$
        $result \leftarrow result * (1 - S_1[r,i] * V[j])$;
        remove $I[0]$ from I;
        remove $J[0]$ from J;
return $1 - result$

by skipping over all zero-values, we reduce POA to $O(T+K)$ operations. Getting the nonzero indices of V is easy with $numpy.flatnonzero$ (this value can even be set-aside for other r's), and for $S_1[r,:]$, extracting the non-zero indices is trivial.