EPFL

HPC for numerical methods and data analysis

Fall Semester 2023

Prof. Laura Grigori

Assistant: Mariana Martinez

Session 11 – November 28, 2023

Randomized rank revealing factorizations for low rank approximation

Exercise 1: Column selection with randomized QRCP

The truncated SVD provides the best low rank approximation in terms of the Frobenius and L2 norms. Sometimes we don't want to compute the full SVD because it might be expensive to do so. Last week we implemented a deterministic rank revealing factorization using strong RRQR. We were able to detect columns of A, I_{02} from which to construct a low rank approximation.

This was based on the fact that for a given matrix $A \in \mathbb{R}^{m\times n}$ there is a permutation P_c and an integer k such that the QR factorization with column pivoting:

$$AP_c = QR = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix} \begin{bmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{bmatrix}$$

reveals the numerical rank k of A. The upper $k \times k$ triangular matrix R_{11} is well conditioned, $\|R_{22}\|_2$ is small and R_{12} is linearly dependent on R_{11} with coefficients bounded by a low-degree polynomial in n. In our case we used a binary tree of depth $\log_2(n/k)$. This gives us the following bound:

$$\|R_{11}^{-1}R_{12}\|_{\max} \leq \frac{1}{\sqrt{2k}} \left(\frac{n}{k}\right)^{\log_2(\sqrt{2}fk)}.$$

Notice that if this is the case then we can build a low rank approximation to A as follows:

$$\tilde{A}_{qr} = Q_1 \begin{bmatrix} R_{11} & R_{12} \end{bmatrix} P_c^\top = Q_1 Q_1^\top A.$$

We have the following bounds for the singular values:

$$1 \leq \frac{\sigma_i(A)}{\sigma_i(R_11)}, \frac{\sigma_j(R_{22})}{\sigma_{k+j}(A)} \leq \sqrt{1 + f^2 k(n-k)}.$$

(Note: if you are confused with what f is refer to last week's exercises or to the lecture notes)

The downside of this algorithm is that is (much) more expensive than regular QR factorization without column pivoting. It has been shown that their randomized counterparts, RQRCP can be

as reliable with failure probabilities exponentially decaying in oversampling size. This week we are going to implement a rather simple version of RQRCP based on last week's code.

The idea is as follows:

Algorithm 1 RQRCP

Input: $A \in \mathbb{R}^{m\times n}$, $\Omega \in \mathbb{R}^{l\times m}$, k $l > k$
Output: I_{02}, indices of the columns of A from which to build the low rank approximation
Compute $B = \Omega A$, $B \in \mathbb{R}^{l\times n}$.
Compute k steps of QRCP on B and select k columns.
Return k selected columns, with indices saved in I_{02}.

With this setup with have the following bounds for $1 \leq j \leq k$:

$$\sigma_j^2(A) \leq \sigma_j^2\left(\begin{bmatrix} R_{11} & R_{12}\end{bmatrix}\right) + \|R_{22}\|_2^2 \tag{1}$$

$$\|R_{22}\|_2 \leq g_1 g_2 \sqrt{(l+1)(n-l)}\sigma_{l+1}(A) \tag{2}$$

where:

$$g_1 \leq \sqrt{\frac{1+\varepsilon}{1-\varepsilon}}$$

$$g_2 \leq \frac{\sqrt{2(1+\varepsilon)}}{1-\varepsilon}\left(1+\sqrt{\frac{1+\varepsilon}{1-\varepsilon}}\right)^{l-1}$$

$$\varepsilon \in (0,1)$$

$$l - k \geq \lceil \frac{4}{\varepsilon^2}\log\left(\frac{2nk}{\delta}\right)\rceil - 1$$

For more about this, check Xiao, Gu, and Langou's paper *Fast Parallel Randomized QR with Column Pivoting Algorithms for Reliable Low-rank Matrix Approximations.*

a) Consider a matrix A partitioned into 4 column blocks. Each processor has one of these blocks.

b) Implement RQRCP using your code from last week.

c) Test your method with two different matrices and different values of l (keep k fixed):

- $A = H_n D H_n^\top$, where H_n is the normalized Hadamard matrix of dimension n, D is a diagonal matrix of your choice. Pick n to be "small".
- Load the normalized MNIST data set and build A as in the project (or last week's exercises). Select a few columns and rows.

d) Comment your results with the different matrices. Do you notice any significant differences with deterministic QRCP?

e) Build a low rank approximation of A. Check the L2 norm of the error with respect to the error of the truncated SVD.

f) Check if the singular values of these selected columns approximate well the singular values of A.

g) Check if the diagonal elements of R_{11} approximate well the singular values of A.

h) Check the bounds 1 and 2.

Using last week's code and implementing the randomization we get:

```
from mpi4py import MPI
import numpy as np
from numpy.linalg import norm, svd
import matplotlib.pyplot as plt
from copy import deepcopy
from scipy.linalg import hadamard, qr
from math import sqrt, exp, ceil, log
import pandas as pd

# Custom library
from sRRQR import sRRQR_rank

plt.ion()

# Functions to build the matrix A (see last week's exercises)
def readData(filename, size = 784, save = False):
    '''
    Read MNIST sparse data from filename
    and transforms this into a dense
    matrix, each line representing an entry
    of the database (i.e. a "flattened" image)
    '''
    dataR = pd.read_csv(filename, sep=',', header = None)
    n = len(dataR)
    data = np.zeros((n, size))
    labels = np.zeros((n, 1))
    # Format accordingly
    for i in range(n):
        l = dataR.iloc[i, 0]
        labels[i] = int(l[0]) # We know that the first digit is the label
        l = l[2:]
        indices_values = [tuple(map(float, pair.split(':'))) for pair in l.split()]
        # Separate indices and values
        indices, values = zip(*indices_values)
        indices = [int(i) for i in indices]
        # Fill in the values at the specified indices
        data[i, indices] = values
    if save:
        data.tofile('./denseData.csv', sep = ',',format='%10.f')
        labels.tofile('./labels.csv', sep = ',',format='%10.f')
    return data, labels

def buildA_sequential(data, c = 1e3, save = False):
    '''
    Function to build A out of a data base
    using the RBF exp( -||x_i - x_j||/c)
    Notice that we only need to fill in the
    upper triangle part of A since it's symmetric
    and its diagonal elements are all 1.
```

```
    '''
    n = data.shape[0]
    A = np.zeros((n, n))
    for j in range(n):
        for i in range(j):
            A[i,j] = exp( -norm( data[i, :] - data[j, :])**2/c)
    A = A + np.transpose(A)
    np.fill_diagonal(A, 1.0)
    if save:
        A.tofile('./A.csv',sep=',',format='%10.f')
    return A

def matTranspose(A, n):
    '''
    Since we want to distribute A's columns
    we need to manipulate our data into the correct
    order before sending it.
    '''
    arrs = np.split(A, n, axis = 1)
    raveled = [np.ravel(arr) for arr in arrs]
    A_transpose = np.concatenate(raveled)
    return A_transpose

# Initialize MPI (world)

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

A = None
m = None
n = None
l = None
local_sizeCols = None
B_T = None
f = 1.3
k = 9
eps = 0.5
Δ = 0.75

if rank == 0:
    # Read the files and build the matrix A here (see last week's exercises)
    filename = "mnist_780"
    data, labels = readData(filename)
    A = buildA_sequential(data)
    A = A[:, 0:200]
    # D = np.diag([1, 0.9, 0.8, 0.7, 0.65, 0.4, 0.1, 0.001])
    # H = hadamard(8)
    # H = 1/norm(H[:, 0])*H
    # A = H@D@np.transpose(H)
    # A = np.arange(0, 108)
    # A = np.reshape(A, (9, 12)) + 0.0
    m = A.shape[0]
    n = A.shape[1]
    # Set the oversampling size (change this and see what happens)
    l = int( k - 1 + ceil(4/(eps**2)*log(2*n*k/Δ) ) )
    print("m: ", m, "n: ", n, "l: ", l)
    # Get the sketching matrix
    Omega = np.random.normal(scale=1/l, size = (l, m))
```

```
    # Get B
    B = Omega@A
    _, S, _ = svd(B)
    print("largest singular value B: ", S[0], " smallest: ", S[-1])
    print("B shape: ", B.shape)
    local_sizeCols = n//size # number of columns
    B_T = matTranspose(B, n)

# Step one: partition A into 4 column blocks
local_sizeCols = comm.bcast(local_sizeCols, root=0)
l = comm.bcast(l, root=0)
B_local = np.empty((local_sizeCols, l))
comm.Scatter(B_T, B_local, root = 0)
# Step 0
# From each column block A1i, i = 1, . . . , 4, k columns are selected by using
# strong RRQR, and their indices are given in Ii0.
Q, R, P = sRRQR_rank(np.transpose(B_local), f, k)
P = P[0:k]
Pall = deepcopy(P) + rank*local_sizeCols
#Start iterating
for i in range(1, ceil(log(size)) + 1):
    #print("i: ", i)
    if rank%(2**i) == 0 and rank + 2**(i-1)<size:
        j = rank + 2**(i-1)
        Phere = np.copy(P)
        Psave = deepcopy(Pall)
        Bhere = np.copy(B_local)
        Bhere = Bhere[Phere, :]
        # We receive the first k indices from Pj from processor j
        #print("Receiving and factorizing at: ", rank)
        comm.Recv(P, source = j, tag = 77)
        comm.Recv(B_local, source = j, tag = 88)
        comm.Recv(Pall, source = j, tag = 99)
        B_local = B_local[P, :]
        B_local = np.concatenate((Bhere, B_local))
        Pall = np.concatenate((Psave, Pall))
        # strong RRQR
        Q, R, P = sRRQR_rank(np.transpose(B_local), f, k)
        P = P[0:k]
        Pall = Pall[P]
    elif rank%(2**i) == 2**(i-1):
        # Send the local P
        #print("Sending from: ", rank)
        comm.Send(P, dest = rank - 2**(i-1), tag = 77)
        comm.Send(B_local, dest = rank - 2**(i-1), tag = 88)
        comm.Send(Pall, dest = rank - 2**(i-1), tag = 99)


if rank == 0:
    # Print the selected columns I02
    # Make the low rank approximation
    print("\nRandomized QRCP\n")
    Pall = Pall.flatten()
    restCols = [i for i in range(n) if i not in Pall]
    orderCols = list(Pall) + restCols
    print("Selected columns: ", Pall)
    Q, R = qr(A[:, orderCols])
    R11 = R[0:l, 0:l]
    U, Sigma, V = svd(R[0:l, :])
```

```
    sigmaR = Sigma[0:k]
    Q1 = Q[:, 0:l]
    R22 = R[l:, l:]
    print("size R22: ", R22.shape)
    # Compute the rhs of the bounds
    g1 = sqrt((1+eps)/(1-eps))
    g2 = sqrt(2 + 2*eps)/(1-eps)*(1 + sqrt((1+eps)/(1-eps)))**(l-1)
    # Get the singular values from the diagonal of R
    print("Approximated singular values (first k of them): ")
    print(sigmaR)
    # Singular values of full A
    Uf, Sigma, Vf = svd(A)
    Uf = Uf[:, 0:k]
    Sigmaf = Sigma[0:k]
    Vf = Vf[0:k, :]
    appfull = Uf@np.diag(Sigmaf)@Vf
    # Get the full singular values
    print("Exact singular values (first k of them): ")
    print(Sigmaf)
    # Get bound 1
    print("|sigma_j(A) - sigma_j(R)|/(sigma_j(A)): ",
          (np.power(Sigmaf, 2) - np.power(sigmaR, 2))/(np.power(Sigmaf, 2)) )
    # Get bound 2
    print("Norm of R22:    ", norm(R[k:, k:]) )
    print("LHS of bound 2: ", g1*g2*sqrt((l+1)*(n-l))*Sigma[l+1] )
    # Compute the L2 error with respect to the SVD low rank approximation
    appsRQRCP = Q1@np.transpose(Q1)@A
    print("L2 error with respect to truncated SVD: ",
          norm( appsRQRCP - appfull)  )
    print("L2 error with respect to full A: ",
          norm(appsRQRCP - A))
```