Project 1: Image Compression

Nima Leclerc , Luther Lu
CS 4220, Numerical Analysis: Linear and Nonlinear Problems
College of Engineering, Cornell University, Ithaca, New York

March 3, 2018

*

1 Introduction

For this project, we are going to work with images, and a simple form of so-called matrix completion. One note up front, we have picked an application (this type of algorithms has broad ranging applications beyond imaging) and algorithms that you have all of the tools to complete. The state of the art in this field involves more complicated algorithms, and for images in particular there are modifications that can be made to improve the results. If you are curious about this area I would be happy to chat more about it. For the purposes of this project, all of the images we will work with are gray-scale and are represented as matrices with real valued entries between 0 and 1 (0 for black and 1 for white). So, an image that is n1 n2 pixels will be represented by an n1 n2 matrix. These techniques can be expanded to deal with color images, but here we will focus on the linear algebra. When producing plots/images (e.g. with the imagesc command in Matlab) you may want to set the color space to gray so that it looks as you would expect. In Matlab this can be accomplished via the command colorspace gray *

2 Methodology

The algorithm implemented in this problem involves the Singular Value Decomposition (SVD) of a matrix A representing the pixel values of an image with $n_1 \times n_2$ pixels. The overarching goal of this method is to approximate (represent) a large matrix as an approximation in k dimensions. This can be accomplished by decomposing A into the following

$$A = USV^T$$

In which case, both U and V are orthogonal matrices of dimensions $n_1 \times p$ and $n_2 \times p$ for $p = min\{n_1, n_1\}$. S is a $n_1 \times p$ matrix, which consists of the singular values of A along the diagonal. However, if one only considers the first k columns of U, the factorization can be approximated as $A_k \approx U_k S_k V_k^T$. This approximation is what will be considered in this image compression algorithm. Now one seeks to minimize the the following norm, hence minimizing the discrepancy between a desired matrix A and the SVD approximation to it, $||A - U_k S_k V_k^T||_F$. Similarly, this is done with the Frobenius Norm. One S_k containing the singular values of the $k \times k$ block matrix. If the singular values of A decay rapidly there may be a good low rank approximation for A, and if k is small enough, storing U_k, V_k, and S_k may be cheaper than storing A. k This can now be written as an approximation of A as Z^T where $W \in R^{n_1 \times k}$ and $Z \in R^{n_2 \times k}$. The low-rank approximation problem can then be cast as the optimization problem, with $A \approx U_k S_k V_k^T = WZ^T$. Hence, in this problem one seeks too minimize the error : $||A - WZ^T||$ with respect to the Z and W matrices. Such an optimization problem can be achieved by first performing a QR factorization to obtain the matrices Q and R necessary to solve the system below

$$Q^T A = RZ^T$$

Then, a similar decomposition can be carried out for Z^T, $Z^T = QR$. The Gram-Schmidt Algorithm will be implemented here. The algorithm is designed to take in an $n \times k$ matrix A which represents the pixel values of the approximate image. It will be a general matrix here however. Now, one seeks an orthogonal matrix Q and upper triangular matrix R such that $QR = A$. The algorithm below depicts this procedure. Let $a_{i,j}$ denote the i, jth element of matrix A.

This was implemented on a random matrix A and achieved the scaling depicted in Figure 1. The implemented algorithm is shown below. Factorization was performed on the random 5×4 matrix as shown here. f

Algorithm 1 Gram-Schmidt Orthogonalization Algorithm

1: for $j = 1 : n$
2: $q_j = a_j$
3: for $i = 1 : n$
4: $r_{ij} = q_j \dot{q}_i$
5: end
6: $r_{jj} = q_j / r_{jj}$

```
%% Gram Schmidt Algorithm
%% Input: mxn matrix, A
%% Output: mxm Q and mxn R , orthonormal and upper triangular matrices
function [Q, R]= QR_FACT(A)
[m,n]=size(A);
Q =zeros(m,n);
R=zeros(n,m);
tic
    for j=1:n %% Loop through all columns of A
         v=A(:,j);
        for i=1:j-1
            R(i,j)=Q(:,i)'*A(:,j); %% Compute R elements
            v=v-R(i,j)*Q(:,i);
        end
    R(j,j)=-norm(v);
    Q(:,j)=v/R(j,j);
    end
end
```

$$A = \begin{bmatrix} 0.8807 & 0.1813 & 0.0731 & 0.2397 \\ 0.7444 & 0.9466 & 0.1946 & 0.9595 \\ 0.4168 & 0.1008 & 0.4175 & 0.3055 \\ 0.9074 & 0.3880 & 0.2929 & 0.1549 \\ 0.0943 & 0.2892 & 0.7021 & 0.5555 \end{bmatrix}$$

Comparing with the algorithm in the MATLAB qr function, we obtained

$$QR_{qr} = \begin{bmatrix} -0.5763 & 0.4448 & 0.1256 & 0.4955 & 0.4569 \\ -0.4871 & -0.7868 & 0.3061 & 0.1821 & -0.1300 \\ -0.2727 & 0.1884 & -0.4640 & 0.3265 & 0.7538 \\ -0.5937 & 0.1633 & -0.0748 & -0.7835 & -0.0376 \\ -0.0617 & -0.3478 & -0.8183 & 0.0307 & 0.4524 \end{bmatrix} \begin{bmatrix} -1.5283 & -0.8413 & -0.4680 & -0.8150 \\ 0 & -0.6823 & -0.2383 & -0.7586 \\ 0 & 0 & -0.7215 & -0.2841 \\ 0 & 0 & 0 & 0.28 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

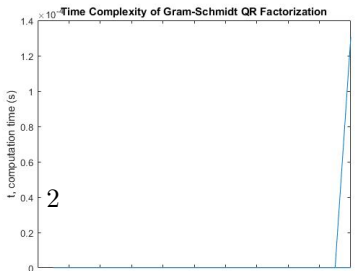
Using, our Gram-Schmidt QR algorithm, the following was obtained

$$QR_{GS} = \begin{bmatrix} -0.5763 & 0.4448 & 0.1256 & 0.4955 & 0.4569 \\ -0.4871 & -0.7868 & 0.3061 & 0.1821 & -0.1300 \\ -0.2727 & 0.1884 & -0.4640 & 0.3265 & 0.7538 \\ -0.5937 & 0.1633 & -0.0748 & -0.7835 & -0.0376 \\ -0.0617 & -0.3478 & -0.8183 & 0.0307 & 0.4524 \end{bmatrix} \begin{bmatrix} -1.5283 & -0.8413 & -0.4680 & -0.8150 \\ 0 & -0.6823 & -0.2383 & -0.7586 \\ 0 & 0 & -0.7215 & -0.2841 \\ 0 & 0 & 0 & 0.28 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Hence, it is clear that our algorithm is in excellent agreement with the one implemented by the qr function in MATLAB. We achieved the scaling depicted in Figure 1. This computation was done on a random 5×4 matrix. The figure suggests a linear scaling with number of operations, which is expected given the small size of this matrix. The alternating least squares algorithm goes as follows,

Such an algorithm can be implemented using the Frobenius norm. This norm can be computed as,

$$||A - WZ^T||_F^2 = \sum_i \sum_j A_{ij}^2 - W_{ij}^2 (Z_{ij}^T)^2)$$



Algorithm 2 Alternating least squares, without regularization

initialize $W \in R^{n_1 \times k}, Z \in R^{n_2 \times k}$
2: Z $\leftarrow argmin_Z ||A - WZ^T||_F^2$
W $\leftarrow argmin_W ||A - WZ^T||_F^2$

Initially, one seeks the factorization of W. This is done by first assuming a matrix W. So, $W = QR$. The above would just be written as,

$$= \sum_i \sum_j A_{ij}^2 - (Q_{ij}R_{ij})^2 (Z_{ij}^T)^2$$

The least squares problem for W then becomes (sum over all columns),

$$\min_Z \sum_{j=1}^{n_1} ||A_j - QRZ^T||^2$$

The minimum value is attained when the argument of the above is 0. Hence, one weeks a solution to the below expression for the ith row of Z . Note that W is an $n_1 \times k$ matrix and Z is an $n_2 \times k$ matrix.

$$Q^T A_i = RZ_i^T$$

Solving this for every column of Z (up to the kth column) will yield Z. Similarly, this can be done for W. The same approach will be used in finding the minimizing W value, however here the other parameter Z has been optimized. Using the previously computed Z, W is factorized as $Z^T = QR$ (using the implemented GS algorithm) and the following least squares problem is solved,

$$\min_W \sum_{i=1}^{n_2} ||A_i - W_i QR||^2$$

Hence, a solution to the following is sought after

$$A_i R^T = W_i Q$$

Performing this operation up to the kth row will yield the minimizing matrix W. Hence, this algorithm collectively yields optimal W and Z matrices for a given k approximation. The product of these (WZ^T) yields the k approximation to A. The Alternating Least Squares Algorithm was implemented in MATLAB as shown below,

```
%% Alternating Least Squares Algorithm
%% INPUT: mxn matrix A, approximation number k
% OUTPUT: m x k and kxm matrices ZT and W

function [ZT, W,approx]= ALS(A, k)
[m,n] = size(A);
W=eye(m,k); %Initialize W matrix as an identity matrix
[q, r] = QR_FACT(W); %Implement QR factorization
ZT=zeros(k,n); %Initialize Z matrix
c=0;
while d<20 %% Loop through 20 times
  for j=1:m
       ZT(:,j) = r/(q'*A(:,j)'); %Compute columns transpose of Z yielding minimization
    for i = 1:n
       [q,r] = QR_FACT(ZT); %Compute QR factorization on transpose of Z
       W(i,:) = (A(i, :)*q)/r'; % Compute rows of optimal W matrix
    end
  end
 d = d+1;
 approx = ZT*W; %Return approximation to A
end
```

This implementation was then tested on the Cornell University crest to a $k = 75$ approximation.

We will now explore the case when not all pixel entries of the the image are known. In the preceding section, alternating least squares seems like a roundabout way to get at something we know how to compute. But now we will explore a setting where we cannot compute the SVD as we might like. Specifically, we will consider being given an image A where we only know the true values of some subset of the pixels, the rest are unobserved. In this case we can still use a variant of alternating least squares to try and recover an underlying low-rank approximation that should roughly resemble our image.

The two key differences are that we will add regularization and only measure error on the observed pixels. Let Ω be a set of (i, j) pairs that represents the set of pixels we observe in the image. So, for example, if $\Omega = \{(1, 2), (10, 20)\}$ then we are only observing the $(1, 2)$ and $(10, 20)$ pixels (matrix entries) of the image. We now define a restricted variant of the Frobenius norm as follows

$$\|A - B\|_{F,\Omega}^2 = \sum_{(i,j)\in\Omega} (A_{i,j} - B_{i,j})^2 .$$

This norm simply measures the difference between A and B on a subset of their entries defined by Ω. The second adjustment we will make is to not allow W or Z to get to large in a manner that depends on some parameter β. This yields the following optimization problem

$$\min_{W,Z} \left\|A - WZ^T\right\|_{F,\Omega}^2 + \beta^2 \|W\|_F^2 + \beta^2 \|Z\|_F^2 .$$

The hope is that WZ^T is then a good approximation of the underlying image. We can now rephrase alternating least squares with regularization and incomplete information in Algorithm 3, once again we will simply run it for a small number of iterations rather than checking for convergence as one would in practice. One can now consider the

Algorithm 3 Alternating least squares, with regularization and unknown entries

initialize $W \in R^{n_1 \times k}$, $Z \in R^{n_2 \times k}$
while not converged **do**

$$Z \leftarrow argmin_Z \left\|A - WZ^T\right\|_{F,\Omega}^2 + \beta^2 \|Z\|_F^2$$

$$W \leftarrow argmin_W \left\|A - WZ^T\right\|_{F,\Omega}^2 + \beta^2 \|W\|_F^2$$

end while
return W, Z

case,

$$\min_x \|Ax - b\|_2^2 + \beta^2 \|x\|_2^2$$

This can be rewritten as the following,

$$\min_x \left\| \begin{bmatrix} A \\ \beta I \end{bmatrix} x - \begin{bmatrix} b \\ 0 \end{bmatrix} \right\|_2^2$$

If one writes the first term of the argument of the above as,

$$\sum_i \sum_j (a_{ij} x_i)^2 - {b_i}^2)$$

and the second term as

$$\sum_i \beta^2 x_i^2$$

Then, then the problem can be broken up into 2 parts consisting of the minimization,

$$\min_{x_i} \sum_i \sum_j (a_{ij} x_i)^2 - b_i^2$$

and

$$\min_{x_i} \beta^2 x_i^2$$

In matrix form,

$$\min_x ||Ax - b||_2^2$$

$$\min_x ||\beta I x||_2^2$$

Hence, this can collectively be written as,

$$\min_x \left\| \begin{bmatrix} A \\ \beta I \end{bmatrix} x - \begin{bmatrix} b \\ 0 \end{bmatrix} \right\|_2^2$$

Q.E.D. This can now be used to solve for ALS problem with regularization and unknown entries. We can use the above form to derive an analogous expression for matrices. Considering Algorithm 3, we first seek a matrix Z which will solve the following optimization problem,

$$\min_Z \left\| A - WZ^T \right\|_{F,\Omega}^2 + \beta^2 \left\| Z \right\|_F^2$$

First, one would assume some matrix W, followed by a factorization: $W = QR$. Hence the expression becomes,

$$\min_Z \left\| A - QRZ^T \right\|_{F,\Omega}^2 + \beta^2 \left\| Z \right\|_F^2$$

Using the derived form to solve for the minimizing Z, we can consider columns j as done earlier. The expression then becomes,

$$\min_{Z_j} \left\| \begin{bmatrix} R \\ \beta I \end{bmatrix} Z_j^T - \begin{bmatrix} Q^T A_j \\ 0 \end{bmatrix} \right\|_2^2$$

This minimization requires solving the system,

$$\begin{bmatrix} R \\ \beta I \end{bmatrix} Z_j^T = \begin{bmatrix} Q^T A_j \\ 0 \end{bmatrix}$$

Hence, solving this system for every column j, given the subset of pixels Ω will yield the optimal value of Z. This can similarly be done for$W$. Having solved for optimal Z, it can be used to solve for optimal W. Again, a QR factorization is first performed:$Z^T = QR$.

$$W_j \begin{bmatrix} Z^T \\ \beta I \end{bmatrix} = \begin{bmatrix} A_j \\ 0 \end{bmatrix} =$$

$$= W_j \begin{bmatrix} QR \\ \beta I \end{bmatrix}$$

Hence, one seeks to solve the following system

$$\begin{bmatrix} R^T \\ \beta I \end{bmatrix} W_j = \begin{bmatrix} QA_j \\ 0 \end{bmatrix}$$

Solving system for the collection of W_j will yield the optimal matrix W. The product WZ^T computed would give k approximation to the image characterized by A. Firstly, in approximating images without the entire set of pixels, one must first be given a matrix C that defines the subset Ω of pixels. Such a matrix is called a mask. The existence of a pixel is given by 1 and 0 otherwise. This is incorporated in the earlier least squares problem definition. The parameter β defines the regularizer. Overall, the algorithm predicts the full image from the known pixels in the set Ω. This algorithm was implemented in MATLAB and shown below,

```
%% IMAGEFINDER Code
%% INPUT: mxn matrix A, approximation index k, mask matrix mass, regurization paratmer beta
%%
function [W, ZT, r] = IMAGEFINDER(A, k, mask, beta)
    [m, n] = size(A);
    W = eye(m, k);
    c = 0;
    ZT = eye(k, n);
```

```
    while d<20 %% Loop through 20 times for convergenace
        for j = 1 : n
            [v, M, len]=masking(A(:, j), W, mask(:, j), 1);
            temp = M;
           temp(len+1:len+k, 1:k) = beta*eye(k, k);
            a = v;
            a(leng+1:leng+k) = 0;
            [Q, R] = QR_FACT(temp); %WBI is a matrix such that m>n
            ZT(:, j) = R\ (Q'*a);
        end
        for i = 1 : n
            [v, M, l] = masking(A(i, :), ZT, mask(i, :), 2);
            z = M;
            z(1:k, l+1:l+k) = beta*eye(k,k);
            a = v;
            a(:, l+1 : l+k) = 0;

           [Q, R] = QR_FACT(z');
            W(i, :) = (a*Q)/R';
        end
        d = d + 1;
    end
    r = norm(W*ZT-A); %RESIDUAL
end


function [v, M, l] = masking(v, M, mask, d) %% function creates mask
    pos = find(mask == 0);
    if d == 1
        v(pos) = [];
        M(pos, :) = [];
    else
        v(pos) = [];
        M(:, pos) = [];
    end
    l = length(v);
end
```

Note that the function masking above serves to create the mask, resembling C (or set Ω) to obtain predictions. This algorithm was implemented for the images in the Project1test* file. Again the k approxiamtion was set to 75 and β was set to 0.01. This yielded the Figures 2 and 3.

Figure 2: *IMAGEFINDER Algorithm Implemented.* $k = 75, \beta = 0.01$

Figures (1) ad (2) suggests that algorithm was successful in predicting the rest of the pixels from a small subset of pixels in the C matrices. The grey contrasts were well predicted. Perhaps increasing the value of β could result in more pronounced spots. However, collectively the prediction resembles what is sought after.

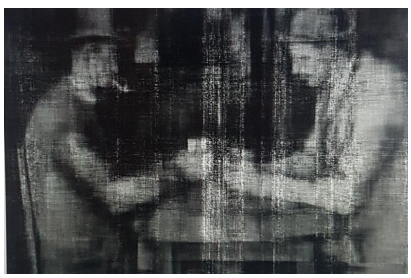


Figure 3: *IMAGEFINDER Algorithm Implemented.*