1a.   
Using = 1.1 for a target value of 6, we observe the following results:

(0.994, [45 452; 0 3]) - This corresponds to an error rate of 1 – 0.994 = 0.006 = 0.6%

1b.  
 Implementing the following to convert x\_train to SVD:

m = size(train\_x, 1)

n\_train = size(train\_x, 3)

n\_test = size(test\_x, 3)

train\_X = Float64.(reshape(train\_x, m\*m, n\_train))

## Set rank for svd

k = 80

U, S, V = svd(train\_X)

x\_approx = U[:,1:k]\*Diagonal(S[1:k])\*V[:,1:k]'

train\_x = reshape(x\_approx, 28, 28, 5000)

println("train\_x converted to SVD Rank 80")

Results in the following: (0.994, [45 452; 0 3]). The error rate remains the same.

1c.  
Implementing the following code to project x\_test onto the rank 80 basis:

## Project test images to SVD Rank 80

test\_X = reshape(test\_x, m\*m, n\_test)

A = U[:, 1:k]

A\_inv = inv(A' \* A)

test\_proj\_x = A \* A\_inv \* A' \* test\_X

test\_x = reshape(test\_proj\_x, m, m, n\_test)

println("test\_x converted to SVD rank 80")

Results in the following: (0.994, [45 452; 0 3]). The error rate remains the same as no SVD approximation.

1d.

Implementing the following code to project x\_train and x\_test onto the NMF rank-80:

## Set rank for nmf

k = 80

## Create input for NMF

m = size(train\_x,1)\*size(train\_x,2)

train\_n = size(train\_x,3)

train\_X = Float64.(reshape(train\_x, m, train\_n)) ## Reshape to 784x5000 matrix

test\_n = size(test\_x,3)

test\_X = Float64.(reshape(test\_x, m, test\_n)) ## Reshape to 784x500 matrix

@time println("Inputs for NMF are defined")

## Convert train images to NMF Rank 80

W, H = NMF.nndsvd(train\_X', k; variant= :ar)

# println(size(W), " ", size(H)) --> Train\_X 784x5000, H = 80x784, W=5000x80

alginst = NMF.ALSPGrad{Float64}(maxiter=100)

r = NMF.solve!(alginst, train\_X', W, H)

@time println("Obtained r for train set")

## reconstruct training set

approx\_x\_train = (r.W \* r.H)'

x\_train = reshape(approx\_x\_train, 28, 28, train\_n)

@time println("train\_x converted to NMF rank 80")

## Project test images into Rank 80 basis

A = r.H'

A\_inv = inv(A' \* A)

approx\_test\_x = A \* A\_inv \* A' \* test\_X

test\_x = reshape(approx\_test\_x, 28, 28, test\_n)

@time println("test\_x projected onto nmf rank 80")

The results are: (0.992, [45 451; 1 3]), indicating an increase in the error rate to 0.8%.

1e.

Based on the results above, I would recommend pursuing the SVD approach further. SVD ensures that the reconstructed images retain a high degree of accuracy. Furthermore, the projection formula used in SVD is straightforward and computationally efficient, which is good for large datasets like MNIST. NMF also offers valuable insights by decomposing data into non-negative factors, but it often requires more complex iterative algorithms and may not capture as much variance within lower ranks. Therefore, for tasks prioritizing accuracy and efficiency at lower ranking bases, SVD is the preferable choice.

2. Using , derive the minimization for in .

Because A is a symmetric, positive definite matrix, we say:

combining the two expressions, we get:

3a.

function solve\_lower\_triangular\_fs(L, b)

# define constants

n = size(L, 1) ## Get number of rows in the n x n matrix L

x = zeros(n) ## Initialize the solution vector x

for i in 1:n ## Row traversal

if L[i, i] == 0 ## Verify that L is invertible

## Raise error if not invertible

error("L is not invertible. No solution for the linear system.")

end

## Solve for x

x[i] = b[i]

println("x[$i]:", x[i])

for j in 1:i-1

x[i] = x[i] - (L[i,j] \* x[j])

end

x[i] = (1 ./ L)[i,i]\*x[i]

end

return x

end

## Test function to verify

L = [2 0 0;

4 -2 0;

-5 1 3]

b = [6; 10; 1]

solve\_lower\_triangular\_fs(L, b)

3b.

Worst case:

Outer loop runs n times, inner loop runs n - 1 times