1a.

Using σ = 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 $f(x) = \frac{1}{2}x^T Ax - x^T b$, derive the minimization for α in $f\big(y - \alpha g(y)\big)$.

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

$$\nabla f(x) = \frac{\partial f(x)}{\partial x^T} = \frac{\partial}{\partial x^T}\left(\frac{1}{2}x^T Ax - x^T b\right) = \frac{\partial}{\partial x^T}\frac{1}{2}(x^T Ax) - \frac{\partial}{\partial x^T}x^T b$$

$$\Rightarrow \frac{\partial}{\partial x^T}\frac{1}{2}x^T Ax = \frac{1}{2}\left[\left(\frac{\partial}{\partial x^T}x^T\right)Ax + x^T\left(\frac{\partial}{\partial x^T}A\right)x + x^T A\left(\frac{\partial}{\partial x^T}x\right)\right] = \frac{1}{2}[Ax + 0 + x^T Ax^T]$$

$$= \frac{1}{2}(2Ax) = Ax$$

$$\Rightarrow \frac{\partial}{\partial x^T}x^T b = b$$

combining the two expressions, we get: $g(x) = \nabla f(x) = Ax - b \Rightarrow g(y) = \nabla f(y) = Ay - b$

$$f\big(y - \alpha g(y)\big) \Rightarrow \frac{1}{2}\big(y - \alpha g(y)\big)^T A\big(y - \alpha g(y)\big) - \big(y - \alpha g(y)\big)^T b$$

$$\Rightarrow \frac{1}{2}[y^T Ay - 2\alpha y^T Ag(y) + \alpha^2 g(y)^T Ag(y)] - [y^T b - \alpha g(y)^T b]$$

$$\Rightarrow \frac{\partial f\big(y - \alpha g(y)\big)}{\partial \alpha} = -y^T Ag(y) + \alpha g(y)^T Ag(y) + g(y)^T b = 0$$

$$\Rightarrow -y^T Ag(y) + \alpha g(y)Ag(y)^T + b^T g(y) = 0$$

$$\Rightarrow \alpha = \frac{y^T Ag(y) - b^T g(y)}{g(y)Ag(y)^T} = \frac{(Ay^T - b^T)g(y)}{g(y)^T Ag(y)} = \frac{g(y)^T g(y)}{g(y)^T Ag(y)}$$

$$\alpha = \frac{(Ay - b)^T(Ay - b)}{(Ay - b)^T A(Ay - b)}$$

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
$n(n-1) = n^2 - n \Rightarrow O(n) = n^2$