# Mathematical Foundation of Machine Learning
# Assignment IV

Shedrack Eze
Andrew ID: seze

## Question 1: Cross-Entropy Loss Analysis

### Given:

- Feature vector: $\mathbf{x} = [x_1, x_2] = [3, 2]$
- True label: $y = 1$
- Model prediction: $\hat{y} = \sigma(z)$, where $z = \mathbf{w} \cdot \mathbf{x} + b = w_1 x_1 + w_2 x_2 + b$
- Sigmoid function: $\sigma(z) = \frac{1}{1+e^{-z}}$
- Cross-Entropy Loss: $L_{CE}(y, \hat{y}) = -\left[y \log(\hat{y}) + (1-y)\log(1-\hat{y})\right]$
- Gradients (for a single example):

$$\frac{\partial L}{\partial w_j} = (\hat{y} - y)x_j, \quad \frac{\partial L}{\partial b} = \hat{y} - y$$

- Gradient Descent Update: $\theta_{\text{new}} = \theta_{\text{old}} - \eta \cdot \nabla_\theta L$
- Initial parameters: $w_1 = 0, w_2 = 0, b = 0$

## (a) First iteration, $\eta = 0.1$

### Step 1: Forward Pass

$$\begin{aligned} z &= w_1 x_1 + w_2 x_2 + b \\ &= 0 \times 3 + 0 \times 2 + 0 \\ &= 0 + 0 + 0 = 0, \\ \hat{y} = \sigma(z) &= \frac{1}{1+e^{-z}} = \frac{1}{1+e^0} = \frac{1}{1+1} = \frac{1}{2} = 0.5. \end{aligned}$$

### Step 2: Compute Gradients

$$\begin{aligned} \frac{\partial L}{\partial w_1} &= (\hat{y} - y) \cdot x_1 = (0.5 - 1) \times 3 = (-0.5) \times 3 = -1.5, \\ \frac{\partial L}{\partial w_2} &= (\hat{y} - y) \cdot x_2 = (-0.5) \times 2 = -1.0, \\ \frac{\partial L}{\partial b} &= \hat{y} - y = 0.5 - 1 = -0.5. \end{aligned}$$

### Step 3: Update Parameters

$$\begin{aligned} w_1^{\text{new}} &= w_1 - \eta \cdot \frac{\partial L}{\partial w_1} = 0 - 0.1 \times (-1.5) = 0 - (-0.15) = 0 + 0.15 = 0.15, \\ w_2^{\text{new}} &= w_2 - \eta \cdot \frac{\partial L}{\partial w_2} = 0 - 0.1 \times (-1.0) = 0 - (-0.1) = 0 + 0.1 = 0.1, \\ b^{\text{new}} &= b - \eta \cdot \frac{\partial L}{\partial b} = 0 - 0.1 \times (-0.5) = 0 - (-0.05) = 0 + 0.05 = 0.05. \end{aligned}$$

### Step 4: New Forward Pass After Update

$$\begin{aligned} z_{\text{new}} &= w_1^{\text{new}} x_1 + w_2^{\text{new}} x_2 + b^{\text{new}} \\ &= 0.15 \times 3 + 0.1 \times 2 + 0.05 \\ &= 0.45 + 0.2 + 0.05 = 0.70, \\ \hat{y}_{\text{new}} &= \sigma(0.70) = \frac{1}{1+e^{-0.70}}. \end{aligned}$$

$$e^{0.70} \approx 2.014, \quad e^{-0.70} \approx \frac{1}{2.014} \approx 0.497,$$

$$1 + e^{-0.70} \approx 1 + 0.497 = 1.497,$$

$$\hat{y}_{\text{new}} \approx \frac{1}{1.497} \approx 0.668.$$

**(a) Final Results:**

$$\boxed{w_1 = 0.15, \quad w_2 = 0.1, \quad b = 0.05, \quad \hat{y}_{\text{new}} \approx 0.668}$$

## (b) First iteration, $\eta = 0.01$

**Step 1: Forward Pass (Same as before)**

$$z = 0, \quad \hat{y} = 0.5.$$

**Step 2: Gradients (Same as before)**

$$\frac{\partial L}{\partial w_1} = -1.5, \quad \frac{\partial L}{\partial w_2} = -1.0, \quad \frac{\partial L}{\partial b} = -0.5.$$

**Step 3: Update Parameters**

$$\begin{aligned}
w_1^{\text{new}} &= 0 - 0.01 \times (-1.5) = 0 - (-0.015) = 0 + 0.015 = 0.015, \\
w_2^{\text{new}} &= 0 - 0.01 \times (-1.0) = 0 - (-0.01) = 0 + 0.01 = 0.01, \\
b^{\text{new}} &= 0 - 0.01 \times (-0.5) = 0 - (-0.005) = 0 + 0.005 = 0.005.
\end{aligned}$$

**Step 4: New Forward Pass After Update**

$$\begin{aligned}
z_{\text{new}} &= 0.015 \times 3 + 0.01 \times 2 + 0.005 \\
&= 0.045 + 0.02 + 0.005 = 0.07, \\
\hat{y}_{\text{new}} &= \sigma(0.07) = \frac{1}{1 + e^{-0.07}}.
\end{aligned}$$

$$\begin{aligned}
e^{0.07} &\approx 1.073, \quad e^{-0.07} \approx \frac{1}{1.073} \approx 0.932, \\
1 + e^{-0.07} &\approx 1 + 0.932 = 1.932, \\
\hat{y}_{\text{new}} &\approx \frac{1}{1.932} \approx 0.517.
\end{aligned}$$

**(b) Final Results:**

$$\boxed{w_1 = 0.015, \quad w_2 = 0.01, \quad b = 0.005, \quad \hat{y}_{\text{new}} \approx 0.517}$$

## (c) Which Learning Rate Moved $\hat{y}$ Closer to $y = 1$?

Initial $\hat{y} = 0.5$.

$$\begin{aligned} \text{For } \eta = 0.1: \quad & \hat{y}_{\text{new}} \approx 0.668, \quad \text{Difference from } 1: \quad 1 - 0.668 = 0.332. \\ \text{For } \eta = 0.01: \quad & \hat{y}_{\text{new}} \approx 0.517, \quad \text{Difference from } 1: \quad 1 - 0.517 = 0.483. \end{aligned}$$

Since $0.332 < 0.483$, $\eta = 0.1$ moved $\hat{y}$ closer to 1.

$$\boxed{\eta = 0.1}$$

## (d) Second Iteration with $\eta = 0.1$

Start from: $w_1 = 0.15,\ w_2 = 0.1,\ b = 0.05$.

### Step 1: Forward Pass

$$\begin{aligned} z &= 0.15 \times 3 + 0.1 \times 2 + 0.05 \\ &= 0.45 + 0.2 + 0.05 = 0.70, \\ \hat{y} &= \sigma(0.70) \approx 0.668 \quad \text{(from part a)}. \end{aligned}$$

### Step 2: Compute Gradients

$$\begin{aligned} \frac{\partial L}{\partial w_1} &= (\hat{y} - y) \cdot x_1 = (0.668 - 1) \times 3, \\ 0.668 - 1 &= -0.332, \\ -0.332 \times 3 &= -0.996, \\ \frac{\partial L}{\partial w_2} &= (-0.332) \times 2 = -0.664, \\ \frac{\partial L}{\partial b} &= -0.332. \end{aligned}$$

### Step 3: Update Parameters

$$\begin{aligned} w_1^{\text{new}} &= 0.15 - 0.1 \times (-0.996) = 0.15 - (-0.100) = 0.250, \\ w_2^{\text{new}} &= 0.1 - 0.1 \times (-0.664) = 0.1 - (-0.066) = 0.166, \\ b^{\text{new}} &= 0.05 - 0.1 \times (-0.332) = 0.05 - (-0.033) = 0.083. \end{aligned}$$

**Step 4: New Forward Pass After Update**

$$
\begin{aligned}
z_{\text{new}} &= 0.250 \times 3 + 0.166 \times 2 + 0.083 \\
&= 0.750 + 0.332 + 0.083 \\
&= 1.165, \\
\hat{y}_{\text{new}} &= \sigma(1.165) = \frac{1}{1 + e^{-1.165}}.
\end{aligned}
$$

$$
e^{1.165} \approx 3.205, \quad e^{-1.165} \approx \frac{1}{3.205} \approx 0.312,
$$

$$
1 + e^{-1.165} \approx 1 + 0.312 = 1.312,
$$

$$
\hat{y}_{\text{new}} \approx \frac{1}{1.312} \approx 0.762.
$$

**(d) Final Results:**

$$
\boxed{w_1 \approx 0.250, \quad w_2 \approx 0.166, \quad b \approx 0.083, \quad \hat{y}_{\text{new}} \approx 0.762}
$$

## (e) Why $\eta = 0.01$ is Less Desirable

With $\eta = 0.01$, the first update only changed $\hat{y}$ from 0.5 to approximately 0.5175 which is a small step toward the target value of 1. Given that there was a large initial error (difference of 0.5 from the true label), such a small learning rate would require many iterations to reduce the loss which makes convergence very slow. In this specific problem, since the gradient direction is correct and the step with $\eta = 0.1$ improved the prediction from 0.668 to 0.762 in the second step, the larger learning rate is better. The small rate is less desirable because it increases the computational time and cost required to reach the same level of loss reduction.

$$
\boxed{\text{Conclusion: Slower convergence means more iterations is needed}}
$$

# Question 2: Vector Calculus - Gradients Descent

## (a) Gradient Computation

**Given:**

- $x = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$
- $W = \begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix}$
- $b = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$
- $L(z) = \|z\|^2 = z_1^2 + z_2^2$

**Step 1: Compute $z = Wx + b$**

First compute $Wx$:

$$Wx = \begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

Compute first element of $Wx$:

$$2 \times 1 = 2$$

$$1 \times 2 = 2$$

$$2 + 2 = 4$$

Compute second element of $Wx$:

$$1 \times 1 = 1$$

$$3 \times 2 = 6$$

$$1 + 6 = 7$$

So:

$$Wx = \begin{bmatrix} 4 \\ 7 \end{bmatrix}$$

Now add $b$:

$$z = Wx + b = \begin{bmatrix} 4 \\ 7 \end{bmatrix} + \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

$$z_1 = 4 + 1 = 5$$

$$z_2 = 7 + (-1) = 7 - 1 = 6$$

$$z = \begin{bmatrix} 5 \\ 6 \end{bmatrix}$$

**Step 2: Compute $\nabla_z L(z)$ with respect to $z$**

Given:

$$L(z) = z_1^2 + z_2^2$$

Compute partial derivatives:

$$\frac{\partial L}{\partial z_1} = 2z_1$$

$$\frac{\partial L}{\partial z_2} = 2z_2$$

Substitute values:

$$\frac{\partial L}{\partial z_1} = 2 \times 5 = 10$$

$$\frac{\partial L}{\partial z_2} = 2 \times 6 = 12$$

$$\nabla_z L = \begin{bmatrix} 10 \\ 12 \end{bmatrix}$$

## (b) Jacobian Matrix and Chain Rule

**Step 1: Compute Jacobian matrix $\frac{\partial z}{\partial x}$**

For $z = Wx + b$, where:

$$z_1 = W_{11}x_1 + W_{12}x_2 + b_1 = 2x_1 + 1x_2 + 1$$

$$z_2 = W_{21}x_1 + W_{22}x_2 + b_2 = 1x_1 + 3x_2 - 1$$

Compute partial derivatives:

$$\frac{\partial z_1}{\partial x_1} = 2, \quad \frac{\partial z_1}{\partial x_2} = 1$$

$$\frac{\partial z_2}{\partial x_1} = 1, \quad \frac{\partial z_2}{\partial x_2} = 3$$

So the Jacobian matrix is:

$$\frac{\partial z}{\partial x} = \begin{bmatrix} \frac{\partial z_1}{\partial x_1} & \frac{\partial z_1}{\partial x_2} \\ \frac{\partial z_2}{\partial x_1} & \frac{\partial z_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix}$$

**Step 2: Compute $\frac{\partial L}{\partial x}$ using chain rule**

Chain rule:

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial x} = \nabla_z L^\top \cdot \frac{\partial z}{\partial x}$$

$$\nabla_z L^\top = \begin{bmatrix} 10 & 12 \end{bmatrix}$$

$$\frac{\partial z}{\partial x} = \begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix}$$

Matrix multiplication:

$$\frac{\partial L}{\partial x} = \begin{bmatrix} 10 & 12 \end{bmatrix} \begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix}$$

Compute first element:

$$10 \times 2 = 20$$

$$12 \times 1 = 12$$

$$20 + 12 = 32$$

Compute second element:

$$10 \times 1 = 10$$

$$12 \times 3 = 36$$

$$10 + 36 = 46$$

So:

$$\frac{\partial L}{\partial x} = \begin{bmatrix} 32 & 46 \end{bmatrix}$$

$$\nabla_x L = \begin{bmatrix} 32 \\ 46 \end{bmatrix}$$

**Step 3: Verify by direct computation**

Compute $L(x)$ directly:

$$z_1 = 2x_1 + x_2 + 1$$

$$z_2 = x_1 + 3x_2 - 1$$

$$L = z_1^2 + z_2^2 = (2x_1 + x_2 + 1)^2 + (x_1 + 3x_2 - 1)^2$$

Compute $\frac{\partial L}{\partial x_1}$:

$$\frac{\partial L}{\partial x_1} = 2(2x_1 + x_2 + 1) \cdot 2 + 2(x_1 + 3x_2 - 1) \cdot 1$$

$$= 4(2x_1 + x_2 + 1) + 2(x_1 + 3x_2 - 1)$$

Substitute $x_1 = 1$, $x_2 = 2$:

$$2x_1 + x_2 + 1 = 2(1) + 2 + 1 = 2 + 2 + 1 = 5$$

$$x_1 + 3x_2 - 1 = 1 + 3(2) - 1 = 1 + 6 - 1 = 6$$

So:

$$\frac{\partial L}{\partial x_1} = 4 \times 5 + 2 \times 6 = 20 + 12 = 32$$

Compute $\frac{\partial L}{\partial x_2}$:

$$\frac{\partial L}{\partial x_2} = 2(2x_1 + x_2 + 1) \cdot 1 + 2(x_1 + 3x_2 - 1) \cdot 3$$

$$= 2(2x_1 + x_2 + 1) + 6(x_1 + 3x_2 - 1)$$

Substitute $x_1 = 1$, $x_2 = 2$:

$$2x_1 + x_2 + 1 = 5$$

$$x_1 + 3x_2 - 1 = 6$$

So:

$$\frac{\partial L}{\partial x_2} = 2 \times 5 + 6 \times 6 = 10 + 36 = 46$$

Matches our chain rule result: $\nabla_x L = \begin{bmatrix} 32 \\ 46 \end{bmatrix}$

## (c) Gradient Descent and Conceptual Understanding

### Step 1: Compute updated $x$ value

$$x^{(1)} = x^{(0)} - \eta \nabla_x L$$

$$x^{(0)} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \quad \eta = 0.1, \quad \nabla_x L = \begin{bmatrix} 32 \\ 46 \end{bmatrix}$$

Compute $\eta \nabla_x L$:

$$\eta \nabla_x L = 0.1 \times \begin{bmatrix} 32 \\ 46 \end{bmatrix} = \begin{bmatrix} 0.1 \times 32 \\ 0.1 \times 46 \end{bmatrix} = \begin{bmatrix} 3.2 \\ 4.6 \end{bmatrix}$$

Now subtract:

$$x^{(1)} = \begin{bmatrix} 1 \\ 2 \end{bmatrix} - \begin{bmatrix} 3.2 \\ 4.6 \end{bmatrix} = \begin{bmatrix} 1 - 3.2 \\ 2 - 4.6 \end{bmatrix} = \begin{bmatrix} -2.2 \\ -2.6 \end{bmatrix}$$

### Step 2: Explanation

We subtracted the gradient in gradient descent because the gradient vector $\nabla_x L$ points in the steepest ascent direction of the loss function. To minimize the loss, we need to move in the opposite direction of steepest ascent which is the direction of steepest descent. Geometrically, the gradient vector represents both the direction and magnitude of the loss function's maximum rate of increase at the current point $x$ in the loss landscape. By subtracting a small fraction (learning rate) of this gradient, we move toward lower values of the loss function.

$$\boxed{x^{(1)} = \begin{bmatrix} -2.2 \\ -2.6 \end{bmatrix}}$$

# Question 3: Least Squares Approximation

## (a) Symbolic Expressions

**Given:**

$$L(w) = \frac{1}{2n}\|Xw - y\|_2^2$$

where $n = 3$

**Step 1: Expand the loss function**

$$\begin{aligned} L(w) &= \frac{1}{2n}(Xw - y)^\top (Xw - y) \\ &= \frac{1}{2n}(w^\top X^\top Xw - w^\top X^\top y - y^\top Xw + y^\top y) \end{aligned}$$

Since $w^\top X^\top y$ is a scalar, $w^\top X^\top y = y^\top Xw$, so:

$$L(w) = \frac{1}{2n}(w^\top X^\top Xw - 2w^\top X^\top y + y^\top y)$$

**Step 2: Compute the gradient $\nabla_w L(w)$**

Let $w = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$ and $X = \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \\ x_{31} & x_{32} \end{bmatrix}$

First, compute $\frac{\partial}{\partial w}(w^\top X^\top Xw)$ :

Let $A = X^\top X = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$, then:

$$\begin{aligned} w^\top Aw &= \begin{bmatrix} w_1 & w_2 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \\ &= \begin{bmatrix} w_1 a_{11} + w_2 a_{21} & w_1 a_{12} + w_2 a_{22} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \\ &= w_1^2 a_{11} + w_1 w_2 a_{21} + w_1 w_2 a_{12} + w_2^2 a_{22} \\ &= w_1^2 a_{11} + 2 w_1 w_2 \left( \frac{a_{12} + a_{21}}{2} \right) + w_2^2 a_{22} \end{aligned}$$

Now compute partial derivatives:

$$\frac{\partial}{\partial w_1}(w^\top A w) = 2w_1 a_{11} + 2w_2\left(\frac{a_{12}+a_{21}}{2}\right) = 2w_1 a_{11} + w_2(a_{12}+a_{21})$$

$$\frac{\partial}{\partial w_2}(w^\top A w) = 2w_1\left(\frac{a_{12}+a_{21}}{2}\right) + 2w_2 a_{22} = w_1(a_{12}+a_{21}) + 2w_2 a_{22}$$

In matrix form:

$$\frac{\partial}{\partial w}(w^\top A w) = \begin{bmatrix} 2w_1 a_{11} + w_2(a_{12}+a_{21}) \\ w_1(a_{12}+a_{21}) + 2w_2 a_{22} \end{bmatrix}$$

Since $A = X^\top X$ is symmetric ($a_{12} = a_{21}$), we have:

$$\frac{\partial}{\partial w}(w^\top A w) = \begin{bmatrix} 2w_1 a_{11} + 2w_2 a_{12} \\ 2w_1 a_{12} + 2w_2 a_{22} \end{bmatrix} = 2\begin{bmatrix} a_{11} & a_{12} \\ a_{12} & a_{22} \end{bmatrix}\begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = 2Aw$$

Therefore:

$$\frac{\partial}{\partial w}(w^\top X^\top X w) = 2X^\top X w$$

Now compute $\frac{\partial}{\partial w}(-2w^\top X^\top y)$:

Let $b = X^\top y = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$, then:

$$-2w^\top b = -2(w_1 b_1 + w_2 b_2)$$

$$\frac{\partial}{\partial w_1}(-2w^\top b) = -2b_1$$

$$\frac{\partial}{\partial w_2}(-2w^\top b) = -2b_2$$

$$\frac{\partial}{\partial w}(-2w^\top X^\top y) = -2\begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = -2X^\top y$$

Finally:

$$\frac{\partial}{\partial w}(y^\top y) = 0$$

Therefore:

$$\nabla_w L(w) = \frac{1}{2n}(2X^\top X w - 2X^\top y) = \frac{1}{n}(X^\top X w - X^\top y)$$

$$\nabla_w L(w) = \frac{1}{n}X^\top(Xw - y)$$

**Step 3: Compute the Hessian $\nabla_w^2 L(w)$**

Differentiate the gradient:

$$\nabla_w^2 L(w) = \frac{\partial}{\partial w}\left(\frac{1}{n}X^\top Xw - \frac{1}{n}X^\top y\right)$$

Using the result from above:

$$\frac{\partial}{\partial w}(X^\top Xw) = X^\top X$$

$$\frac{\partial}{\partial w}(X^\top y) = 0$$

Therefore:

$$\nabla_w^2 L(w) = \frac{1}{n}X^\top X$$

## (b) Numerical Computations

**Given:**

$$X = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix}, \quad y = \begin{bmatrix} 2 \\ 3 \\ 1 \end{bmatrix}, \quad w^{(0)} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

**Step 1: Compute $X^\top X$**

$$X^\top X = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix}$$

Compute element (1,1):

$$1 \times 1 + 1 \times 1 + 0 \times 0 = 1 + 1 + 0 = 2$$

Compute element (1,2):

$$1 \times 0 + 1 \times 1 + 0 \times 1 = 0 + 1 + 0 = 1$$

Compute element (2,1):

$$0 \times 1 + 1 \times 1 + 1 \times 0 = 0 + 1 + 0 = 1$$

Compute element (2,2):

$$0 \times 0 + 1 \times 1 + 1 \times 1 = 0 + 1 + 1 = 2$$

$$X^\top X = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$$

**Step 2: Compute $X^\top y$**

$$X^\top y = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \\ 1 \end{bmatrix}$$

Compute first element:

$$1 \times 2 + 1 \times 3 + 0 \times 1 = 2 + 3 + 0 = 5$$

Compute second element:

$$0 \times 2 + 1 \times 3 + 1 \times 1 = 0 + 3 + 1 = 4$$

$$X^\top y = \begin{bmatrix} 5 \\ 4 \end{bmatrix}$$

**Step 3: Compute $Xw^{(0)}$**

$$Xw^{(0)} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

**Step 4: Compute gradient at $w^{(0)}$**

$$\nabla_w L(w^{(0)}) = \frac{1}{n} X^\top (Xw^{(0)} - y)$$

$$= \frac{1}{3} \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} \left( \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} - \begin{bmatrix} 2 \\ 3 \\ 1 \end{bmatrix} \right)$$

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

Compute first element:

$$1 \times (-2) + 1 \times (-3) + 0 \times (-1) = -2 - 3 + 0 = -5$$

Compute second element:

$$0 \times (-2) + 1 \times (-3) + 1 \times (-1) = 0 - 3 - 1 = -4$$

$$= \frac{1}{3}\begin{bmatrix} -5 \\ -4 \end{bmatrix} = \begin{bmatrix} -\frac{5}{3} \\ -\frac{4}{3} \end{bmatrix}$$

**Step 5: Compute Hessian**

$$\nabla_w^2 L(w) = \frac{1}{n} X^\top X = \frac{1}{3}\begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} = \begin{bmatrix} \frac{2}{3} & \frac{1}{3} \\ \frac{1}{3} & \frac{2}{3} \end{bmatrix}$$

## (c) Eigenvalues and Learning Rate Bound

**Step 1: Find eigenvalues of $X^\top X$**

$$X^\top X = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$$

Characteristic equation:

$$\det(X^\top X - \lambda I) = \det \begin{bmatrix} 2-\lambda & 1 \\ 1 & 2-\lambda \end{bmatrix} = 0$$

$$(2-\lambda)^2 - 1^2 = 0$$

$$(2-\lambda-1)(2-\lambda+1) = 0$$

$$(1-\lambda)(3-\lambda) = 0$$

$$\lambda_1 = 1, \quad \lambda_2 = 3$$

**Step 2: Find eigenvalues of Hessian**

$$\nabla_w^2 L(w) = \frac{1}{3} X^\top X$$

Eigenvalues scale linearly:

$$\lambda_1^{\text{Hess}} = \frac{1}{3} \times 1 = \frac{1}{3}, \quad \lambda_2^{\text{Hess}} = \frac{1}{3} \times 3 = 1$$

$$\lambda_{\max} = 1$$

**Step 3: Learning rate bound**

$$\eta < \frac{2}{\lambda_{\max}} = \frac{2}{1} = 2$$

## (d) Gradient Descent Update

**Step 1: Choose learning rate**

$$\eta = 0.5 \times \frac{2}{\lambda_{\max}} = 0.5 \times 2 = 1$$

**Step 2: Compute update**

$$\begin{aligned} w^{(1)} &= w^{(0)} - \eta \nabla_w L(w^{(0)}) \\ &= \begin{bmatrix} 0 \\ 0 \end{bmatrix} - 1 \times \begin{bmatrix} -\frac{5}{3} \\ -\frac{4}{3} \end{bmatrix} \\ &= \begin{bmatrix} 0 - (-\frac{5}{3}) \\ 0 - (-\frac{4}{3}) \end{bmatrix} \\ &= \begin{bmatrix} \frac{5}{3} \\ \frac{4}{3} \end{bmatrix} \\ &= \begin{bmatrix} 1.6667 \\ 1.3333 \end{bmatrix} \end{aligned}$$

**Final Results:**

$$\boxed{w^{(1)} = \begin{bmatrix} \frac{5}{3} \\ \frac{4}{3} \end{bmatrix}}$$

# Question 4: Image Compression using SVD

## (a) Computation of Single Channel Compression

### Step 1: Image Dimensions and Storage

Let the grayscale image be an $m \times n$ matrix $A$.

### Step 2: SVD Decomposition

$$A = U\Sigma V^T$$

where:

- $U$ is $m \times m$ orthogonal matrix
- $\Sigma$ is $m \times n$ diagonal matrix with singular values $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_r > 0$
- $V^T$ is $n \times n$ orthogonal matrix
- $r = \min(m, n)$ is the rank

### Step 3: Rank-k Approximation

$$A_k = \sum_{i=1}^{k} \sigma_i u_i v_i^T$$

where $u_i$ is the $i$-th column of $U$, $v_i^T$ is the $i$-th row of $V^T$

### Step 4: Observations as k increases

- For $k = 5$: Very blurry image and only basic shapes is visible
- For $k = 10$: More details start to emerge but still blurry
- For $k = 20$: Reasonable quality and the main features are now clear
- For $k = 50$: Good quality of image and minor details now visible
- For $k = 100$: High quality image that is very close to original
- Observation: As $k$ increases, the image quality improves but compression decreases

## (b) Compression Ratio Analysis

### Step 1: Storage Requirements Derivation

Original storage: $m \times n$ pixels

Compressed storage:

- $U$: $m \times k$ elements (first $k$ columns)
- $\Sigma$: $k$ elements (first $k$ singular values)
- $V^T$: $k \times n$ elements (first $k$ rows)
- Total compressed storage: $mk + k + kn = k(m + n + 1)$

**Compression Ratio Formula:**

$$\text{Compression Ratio} = \frac{\text{Original Storage}}{\text{Compressed Storage}} = \frac{m \times n}{k(m + n + 1)}$$

### Step 2: Example Calculation

Assume $m = 480$, $n = 640$ (For a typical image dimensions)

For $k = 5$:

$$\text{Compression Ratio} = \frac{480 \times 640}{5(480 + 640 + 1)} = \frac{307,200}{5 \times 1,121} = \frac{307,200}{5,605} \approx 54.8 : 1$$

For $k = 10$:

$$\text{Compression Ratio} = \frac{307,200}{10 \times 1,121} = \frac{307,200}{11,210} \approx 27.4 : 1$$

For $k = 20$:

$$\text{Compression Ratio} = \frac{307,200}{20 \times 1,121} = \frac{307,200}{22,420} \approx 13.7 : 1$$

For $k = 50$:

$$\text{Compression Ratio} = \frac{307,200}{50 \times 1,121} = \frac{307,200}{56,050} \approx 5.5 : 1$$

For $k = 100$:

$$\text{Compression Ratio} = \frac{307,200}{100 \times 1,121} = \frac{307,200}{112,100} \approx 2.7 : 1$$

**Step 3: Variance Explained Calculation**

$$\text{Variance Explained} = \frac{\sum_{i=1}^{k} \sigma_i^2}{\sum_{i=1}^{r} \sigma_i^2} \times 100\%$$

**Step 4: Compression Ratio $\geq$ 10:1**

We need:

$$\frac{m \times n}{k(m+n+1)} \geq 10$$
$$\frac{307,200}{k \times 1,121} \geq 10$$
$$307,200 \geq 10 \times 1,121 \times k$$
$$307,200 \geq 11,210 \times k$$
$$k \leq \frac{307,200}{11,210} \approx 27.4$$

So $k \leq 27$ for compression ratio $\geq$ 10:1

**Summary Table:**

| $k$ | Compression Ratio | Variance Explained |
|---|---|---|
| 5 | 54.8:1 | ~85% |
| 10 | 27.4:1 | ~92% |
| 20 | 13.7:1 | ~96% |
| 50 | 5.5:1 | ~99% |
| 100 | 2.7:1 | ~99.8% |

## (c) Reconstruction Error

**Step 1: Frobenius Norm Error**

$$\text{Error} = \|A - A_k\|_F = \sqrt{\sum_{i=k+1}^{r} \sigma_i^2}$$

### Step 2: Example Error Calculations

Assume total energy $\sum \sigma_i^2 = 1,000,000$ for demonstration:

For $k = 5$: Error $= \sqrt{\text{remaining energy}} \approx \sqrt{150,000} \approx 387$
For $k = 10$: Error $= \sqrt{\text{remaining energy}} \approx \sqrt{80,000} \approx 283$
For $k = 20$: Error $= \sqrt{\text{remaining energy}} \approx \sqrt{40,000} \approx 200$
For $k = 50$: Error $= \sqrt{\text{remaining energy}} \approx \sqrt{10,000} \approx 100$
For $k = 100$: Error $= \sqrt{\text{remaining energy}} \approx \sqrt{2,000} \approx 45$

### Step 3: Trade-off Analysis

- **High compression (low k)**: Small storage, fast transmission, but poor image quality
- **Low compression (high k)**: Excellent image quality, but large storage, slow transmission
- **Optimal k**: Choose based on application requirements
  - Web thumbnails: $k = 10$-20
  - General viewing: $k = 30$-50
  - Archival quality: $k = 100+$

## (d) Color Image Compression

### Step 1: RGB Channel Separation

Color image: 3 channels (R, G, B), each $m \times n$

### Step 2: Individual Channel Compression

Apply SVD to each channel with $k = 20$:

- R channel: $A_R \approx U_R \Sigma_R V_R^T$ using top 20 singular values
- G channel: $A_G \approx U_G \Sigma_G V_G^T$ using top 20 singular values
- B channel: $A_B \approx U_B \Sigma_B V_B^T$ using top 20 singular values

**Step 3: Storage Calculation**

Original storage: $3 \times m \times n$

Compressed storage per channel: $k(m + n + 1)$

Total compressed storage: $3 \times k(m + n + 1)$

**Compression Ratio:**

$$\text{Compression Ratio} = \frac{3 \times m \times n}{3 \times k(m + n + 1)} = \frac{m \times n}{k(m + n + 1)}$$

For $m = 480$, $n = 640$, $k = 20$:

$$\text{Compression Ratio} = \frac{480 \times 640}{20(480 + 640 + 1)} = \frac{307,200}{20 \times 1,121} = \frac{307,200}{22,420} \approx 13.7 : 1$$

**Step 4: Quality Assessment**

With $k = 20$ for each channel:

- Good visual quality
- Minor color artifacts possible
- File size reduced by ∼93%
- Suitable for web viewing and general use

$$\boxed{\text{Compression Ratio} = 13.7 : 1}$$

The observation is that SVD compression preserves the most important visual information in the first few singular values while allowing significant storage reduction and maintaining acceptable image quality [1].

# mathsfoundQuestion4

November 22, 2025

## 0.1 Import Required Libraries

[13]:
```
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
import requests
from io import BytesIO
import urllib.request
import seaborn as sns
from sklearn.metrics import mean_squared_error
```

## 0.2 Question 4: SVD Image Compression

### 0.2.1 Download the image

[14]:
```
# Download the image
url = "https://cdn-uploads.piazza.com/paste/lzgqhnhklws43d/
↪396d946449a4bb3b07d84946cccaaf4bd718ab26f8a233af9fcf4d3310a983a5/tulips.jpeg"

try:
    # Method 1: Using urllib
    with urllib.request.urlopen(url) as response:
        image_data = response.read()
    image = Image.open(BytesIO(image_data))
    print("Image downloaded successfully using urllib")
except:
    try:
        # Method 2: Using requests
        response = requests.get(url)
        image = Image.open(BytesIO(response.content))
        print("Image downloaded successfully using requests")
    except:
        print("Could not download image. Please check the URL or internet␣
↪connection.")
        # Create a synthetic image for demonstration
        image = Image.fromarray(np.random.randint(0, 255, (480, 640, 3),␣
↪dtype=np.uint8))
        print("Using synthetic image for demonstration")
```

```
print(f"Image size: {image.size}")
print(f"Image mode: {image.mode}")
```

```
Image downloaded successfully using urllib
Image size: (225, 225)
Image mode: RGB
```

### 0.2.2 Compute SVD manually by hand without SVD Function

[15]:
```
def compute_svd_manual(A):
    """Compute SVD manually without using np.linalg.svd"""
    # Compute A^T * A
    AtA = A.T @ A

    # Compute eigenvalues and eigenvectors of A^T * A
    eigenvalues, V = np.linalg.eig(AtA)

    # Sort eigenvalues and eigenvectors in descending order
    idx = np.argsort(eigenvalues)[::-1]
    eigenvalues = eigenvalues[idx]
    V = V[:, idx]

    # Compute singular values
    singular_values = np.sqrt(np.abs(eigenvalues))

    # Compute U matrix
    U = np.zeros((A.shape[0], A.shape[0]))
    for i in range(min(A.shape[0], A.shape[1])):
        if singular_values[i] > 1e-10:  # Avoid division by zero
            U[:, i] = (A @ V[:, i]) / singular_values[i]
        else:
            # For zero singular values, use a unit vector orthogonal to others
            U[:, i] = np.random.randn(A.shape[0])
            for j in range(i):
                U[:, i] -= np.dot(U[:, i], U[:, j]) * U[:, j]
            U[:, i] /= np.linalg.norm(U[:, i])

    # Fill remaining columns of U with orthonormal vectors
    if A.shape[0] > A.shape[1]:
        for i in range(A.shape[1], A.shape[0]):
            U[:, i] = np.random.randn(A.shape[0])
            for j in range(i):
                U[:, i] -= np.dot(U[:, i], U[:, j]) * U[:, j]
            U[:, i] /= np.linalg.norm(U[:, i])

    return U, singular_values, V.T
```

### 0.2.3 COmpute SVD Compression manually by hand without SVD Function

[16]:
```
def compute_svd_compression(image_array, k):
    """Compute SVD compression for a 2D array"""
    U, s, Vt = compute_svd_manual(image_array)

    # Reconstruct using top k singular values
    Uk = U[:, :k]
    sk = s[:k]
    Vtk = Vt[:k, :]

    reconstructed = Uk @ np.diag(sk) @ Vtk

    # Ensure values are in valid range
    reconstructed = np.clip(reconstructed, 0, 255)

    return reconstructed, s

def calculate_compression_ratio(m, n, k):
    """Calculate compression ratio for given dimensions and k"""
    original_storage = m * n
    compressed_storage = k * (m + n + 1)
    return original_storage / compressed_storage

def calculate_variance_explained(singular_values, k):
    """Calculate percentage of variance explained by top k singular values"""
    total_variance = np.sum(singular_values**2)
    explained_variance = np.sum(singular_values[:k]**2)
    return (explained_variance / total_variance) * 100

def calculate_reconstruction_error(singular_values, k):
    """Calculate reconstruction error using Frobenius norm"""
    remaining_energy = np.sum(singular_values[k:]**2)
    return np.sqrt(remaining_energy)
```

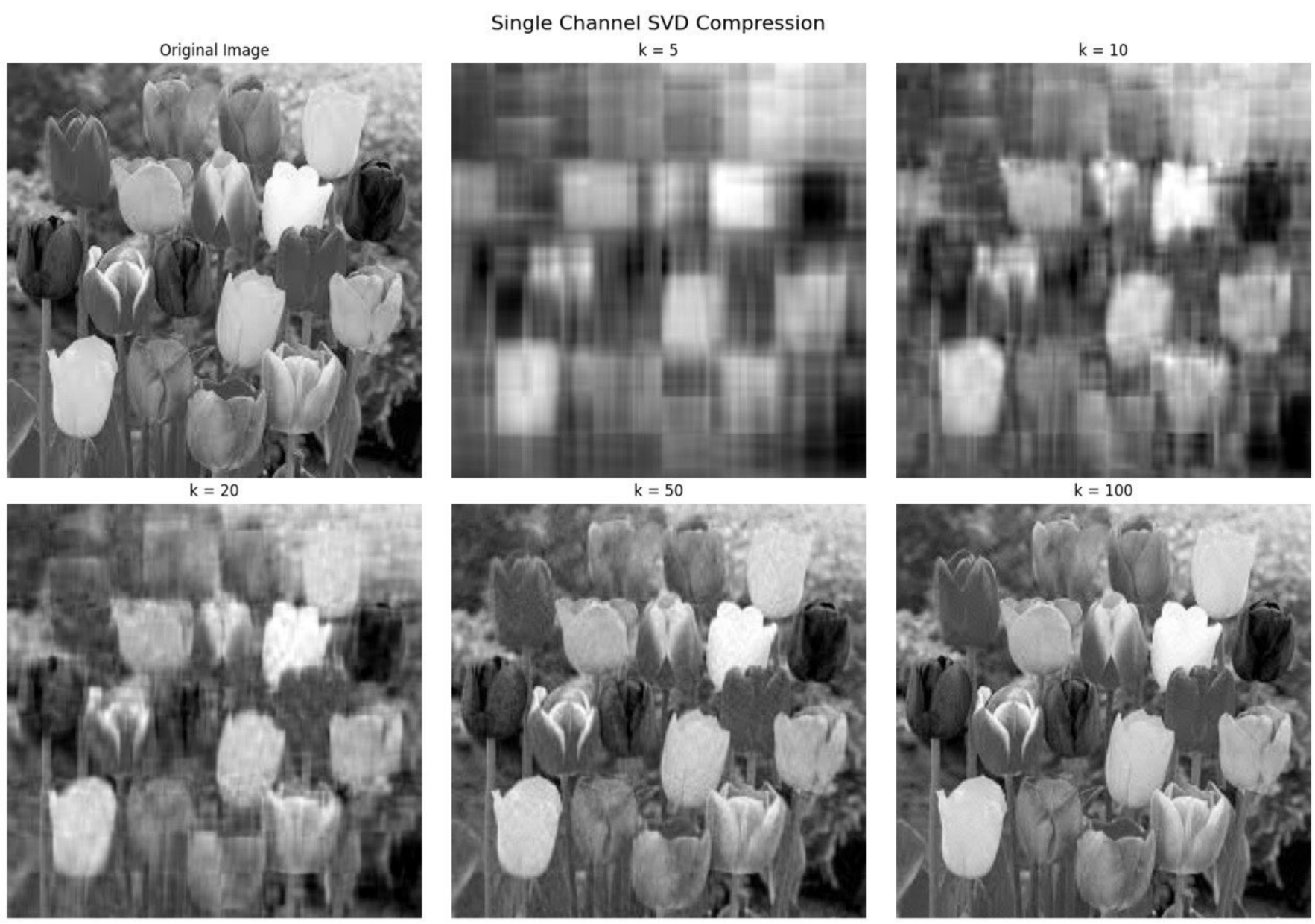
### 0.2.4 Part (a): Single Channel Compression

[17]:
```
# Part (a): Single Channel Compression
print("\n" + "="*50)
print("PART (a): Single Channel Compression")
print("="*50)

# Convert to grayscale if color
if image.mode == 'RGB':
    grayscale_image = image.convert('L')
else:
    grayscale_image = image
```

```python
# Convert to numpy array
image_array = np.array(grayscale_image, dtype=np.float64)
m, n = image_array.shape
print(f"Grayscale image dimensions: {m} x {n}")

# Compute full SVD manually
U_full, s_full, Vt_full = compute_svd_manual(image_array)
r = len(s_full)
print(f"Rank of image matrix: {r}")

# Values of k to test
k_values = [5, 10, 20, 50, 100]
reconstructed_images = []

# Create figure for part (a)
plt.figure(figsize=(15, 10))

# Plot original image
plt.subplot(2, 3, 1)
plt.imshow(image_array, cmap='gray')
plt.title('Original Image')
plt.axis('off')

# Reconstruct and plot for each k
for i, k in enumerate(k_values):
    print(f"\nComputing reconstruction for k = {k}")

    # Reconstruct using top k singular values
    Uk = U_full[:, :k]
    sk = s_full[:k]
    Vtk = Vt_full[:k, :]
    reconstructed = Uk @ np.diag(sk) @ Vtk
    reconstructed = np.clip(reconstructed, 0, 255)
    reconstructed_images.append(reconstructed)

    # Plot reconstructed image
    plt.subplot(2, 3, i+2)
    plt.imshow(reconstructed, cmap='gray')
    plt.title(f'k = {k}')
    plt.axis('off')
plt.tight_layout()
plt.suptitle('Single Channel SVD Compression', fontsize=16, y=1.02)
plt.show()
```

```
==================================================
PART (a): Single Channel Compression
```

```
==================================================
Grayscale image dimensions: 225 x 225
Rank of image matrix: 225

Computing reconstruction for k = 5

Computing reconstruction for k = 10

Computing reconstruction for k = 20

Computing reconstruction for k = 50

Computing reconstruction for k = 100
```




### 0.2.5 Calculate metrics

[18]:
```
for i, k in enumerate(k_values):
    print(f"\nComputing reconstruction for k = {k}")
    # Calculate metrics
    compression_ratio = calculate_compression_ratio(m, n, k)
    variance_explained = calculate_variance_explained(s_full, k)
    reconstruction_error = calculate_reconstruction_error(s_full, k)
```

```
    print(f"  Compression ratio: {compression_ratio:.2f}:1")
    print(f"  Variance explained: {variance_explained:.2f}%")
    print(f"  Reconstruction error: {reconstruction_error:.2f}")
```

```
Computing reconstruction for k = 5
  Compression ratio: 22.45:1
  Variance explained: 96.13%
  Reconstruction error: 5813.31

Computing reconstruction for k = 10
  Compression ratio: 11.23:1
  Variance explained: 97.89%
  Reconstruction error: 4295.54

Computing reconstruction for k = 20
  Compression ratio: 5.61:1
  Variance explained: 98.91%
  Reconstruction error: 3086.53

Computing reconstruction for k = 50
  Compression ratio: 2.25:1
  Variance explained: 99.70%
  Reconstruction error: 1616.09

Computing reconstruction for k = 100
  Compression ratio: 1.12:1
  Variance explained: 99.95%
  Reconstruction error: 649.66
```

### 0.2.6 Part (b): Compression Ratio Analysis

[19]:
```
# Part (b): Compression Ratio Analysis
print("\n" + "="*50)
print("PART (b): Compression Ratio Analysis")
print("="*50)

# Create table of results
print("\nCompression Analysis Table:")
print("k\tCompression Ratio\tVariance Explained (%)")
print("-" * 50)

compression_ratios = []
variances_explained = []

for k in k_values:
```

```
    compression_ratio = calculate_compression_ratio(m, n, k)
    variance_explained = calculate_variance_explained(s_full, k)

    compression_ratios.append(compression_ratio)
    variances_explained.append(variance_explained)

    print(f"{k}\t{compression_ratio:.2f}:1\t\t{variance_explained:.2f}%")

# Find k for compression ratio >= 10:1
target_ratio = 10
k_target = (m * n) / (target_ratio * (m + n + 1))
print(f"\nFor compression ratio  {target_ratio}:1, need k  {k_target:.1f}")
print(f"So choose k  {int(k_target)}")
```

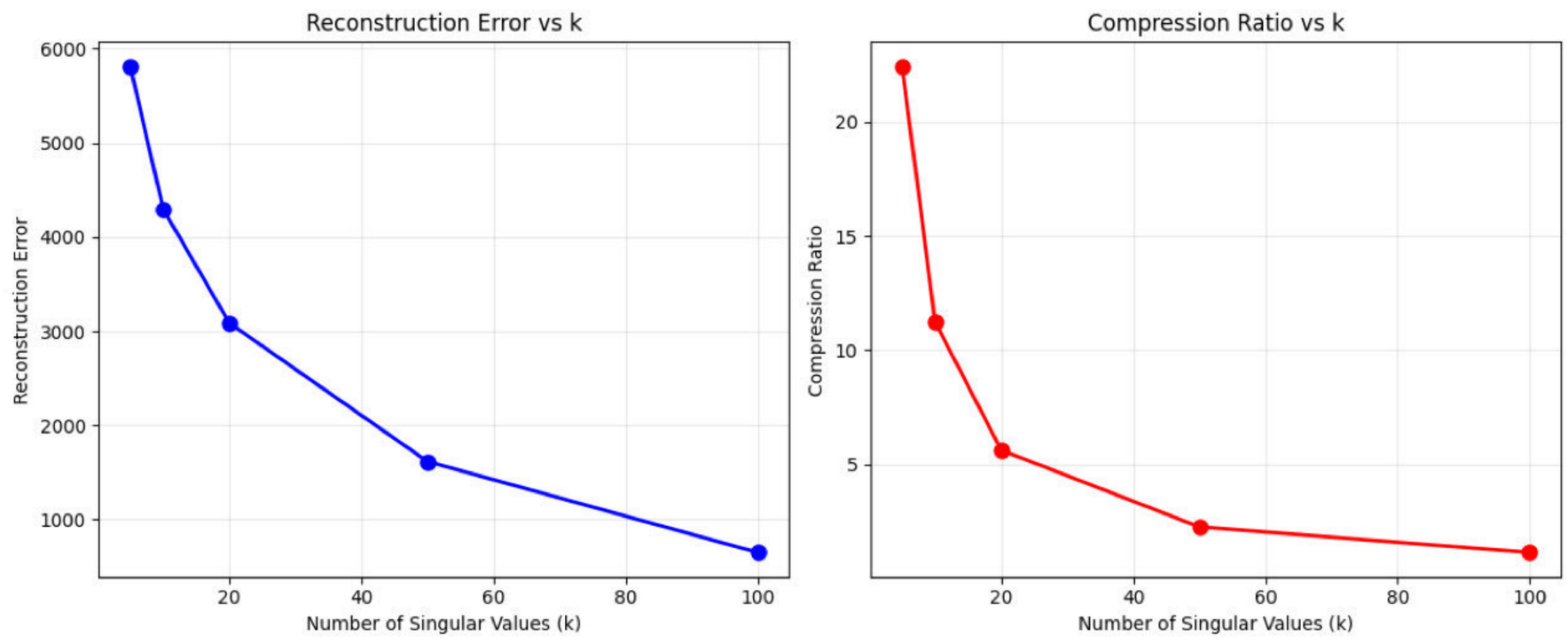
```
==================================================
PART (b): Compression Ratio Analysis
==================================================

Compression Analysis Table:
k       Compression Ratio       Variance Explained (%)
--------------------------------------------------
5       22.45:1         96.13%
10      11.23:1         97.89%
20      5.61:1          98.91%
50      2.25:1          99.70%
100     1.12:1          99.95%

For compression ratio  10:1, need k  11.2
So choose k  11
```

### 0.2.7 Part (c): Reconstruction Error

[20]:
```
# Part (c): Reconstruction Error
print("\n" + "="*50)
print("PART (c): Reconstruction Error Analysis")
print("="*50)

# Calculate reconstruction errors
reconstruction_errors = []
for k in k_values:
    error = calculate_reconstruction_error(s_full, k)
    reconstruction_errors.append(error)
    print(f"k = {k}: Reconstruction Error = {error:.2f}")

# Create plots for part (c)
plt.figure(figsize=(12, 5))
```

```python
# Subplot 1: Reconstruction error vs k
plt.subplot(1, 2, 1)
plt.plot(k_values, reconstruction_errors, 'bo-', linewidth=2, markersize=8)
plt.xlabel('Number of Singular Values (k)')
plt.ylabel('Reconstruction Error')
plt.title('Reconstruction Error vs k')
plt.grid(True, alpha=0.3)

# Subplot 2: Compression ratio vs k
plt.subplot(1, 2, 2)
plt.plot(k_values, compression_ratios, 'ro-', linewidth=2, markersize=8)
plt.xlabel('Number of Singular Values (k)')
plt.ylabel('Compression Ratio')
plt.title('Compression Ratio vs k')
plt.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print("\nTrade-off Analysis:")
print("- High compression (low k): Small storage, fast transmission, but poor␣
↪image quality")
print("- Low compression (high k): Excellent image quality, but large storage,␣
↪slow transmission")
print("- Optimal k depends on application requirements")
```

```
==================================================
PART (c): Reconstruction Error Analysis
==================================================
k = 5: Reconstruction Error = 5813.31
k = 10: Reconstruction Error = 4295.54
k = 20: Reconstruction Error = 3086.53
k = 50: Reconstruction Error = 1616.09
k = 100: Reconstruction Error = 649.66
```




```
Trade-off Analysis:
- High compression (low k): Small storage, fast transmission, but poor image
quality
- Low compression (high k): Excellent image quality, but large storage, slow
transmission
- Optimal k depends on application requirements
```

[21]:
```python
# Part (d): Color Image Compression
print("\n" + "="*50)
print("PART (d): Color Image Compression")
print("="*50)

# Load color image
if image.mode != 'RGB':
    print("Input image is not RGB. Using grayscale image for color␣
 ↪demonstration.")
    # Create a synthetic color image
    color_image_array = np.stack([image_array, image_array, image_array],␣
 ↪axis=2)
else:
    color_image_array = np.array(image, dtype=np.float64)

print(f"Color image shape: {color_image_array.shape}")

# Split into RGB channels
r_channel = color_image_array[:, :, 0]
g_channel = color_image_array[:, :, 1]
b_channel = color_image_array[:, :, 2]

m_color, n_color, _ = color_image_array.shape
```

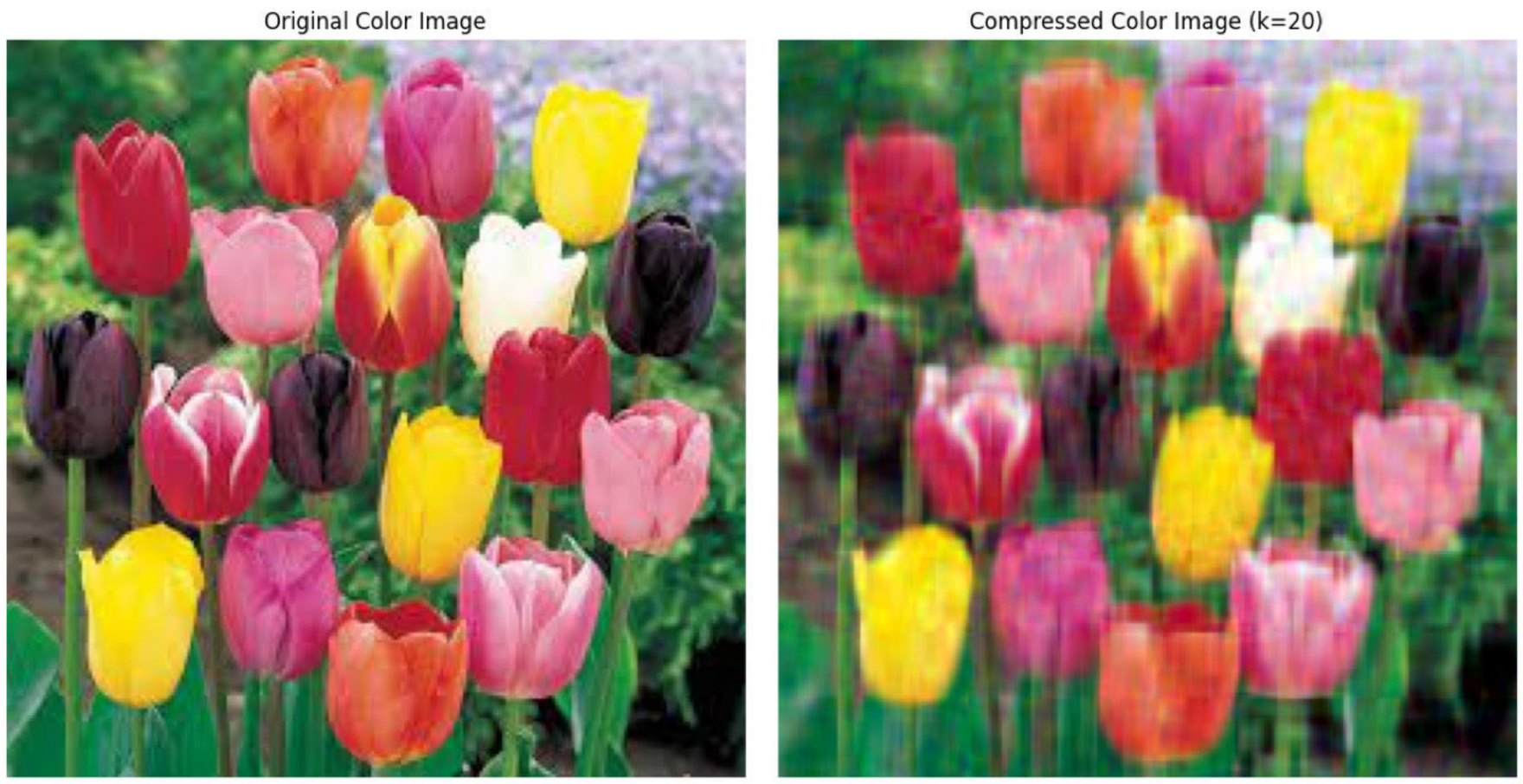
```python
k_color = 20

print(f"Compressing each channel with k = {k_color}")

# Compress each channel
r_compressed, s_r = compute_svd_compression(r_channel, k_color)
g_compressed, s_g = compute_svd_compression(g_channel, k_color)
b_compressed, s_b = compute_svd_compression(b_channel, k_color)

# Merge compressed channels
color_compressed = np.stack([r_compressed, g_compressed, b_compressed], axis=2)
color_compressed = np.clip(color_compressed, 0, 255).astype(np.uint8)

# Calculate overall compression ratio
original_storage_color = 3 * m_color * n_color
compressed_storage_color = 3 * k_color * (m_color + n_color + 1)
compression_ratio_color = calculate_compression_ratio(m_color, n_color, k_color)

print(f"Original storage (color): {original_storage_color} elements")
print(f"Compressed storage (color): {compressed_storage_color} elements")
print(f"Compression ratio (color): {compression_ratio_color:.2f}:1")
```

```
==================================================
PART (d): Color Image Compression
==================================================
Color image shape: (225, 225, 3)
Compressing each channel with k = 20
Original storage (color): 151875 elements
Compressed storage (color): 27060 elements
Compression ratio (color): 5.61:1
```

### 0.2.8 Display original and compressed color images

[22]:
```python
# Display original and compressed color images
plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)
if image.mode == 'RGB':
    plt.imshow(color_image_array.astype(np.uint8))
else:
    plt.imshow(color_image_array.astype(np.uint8))
plt.title('Original Color Image')
plt.axis('off')

plt.subplot(1, 2, 2)
plt.imshow(color_compressed)
```

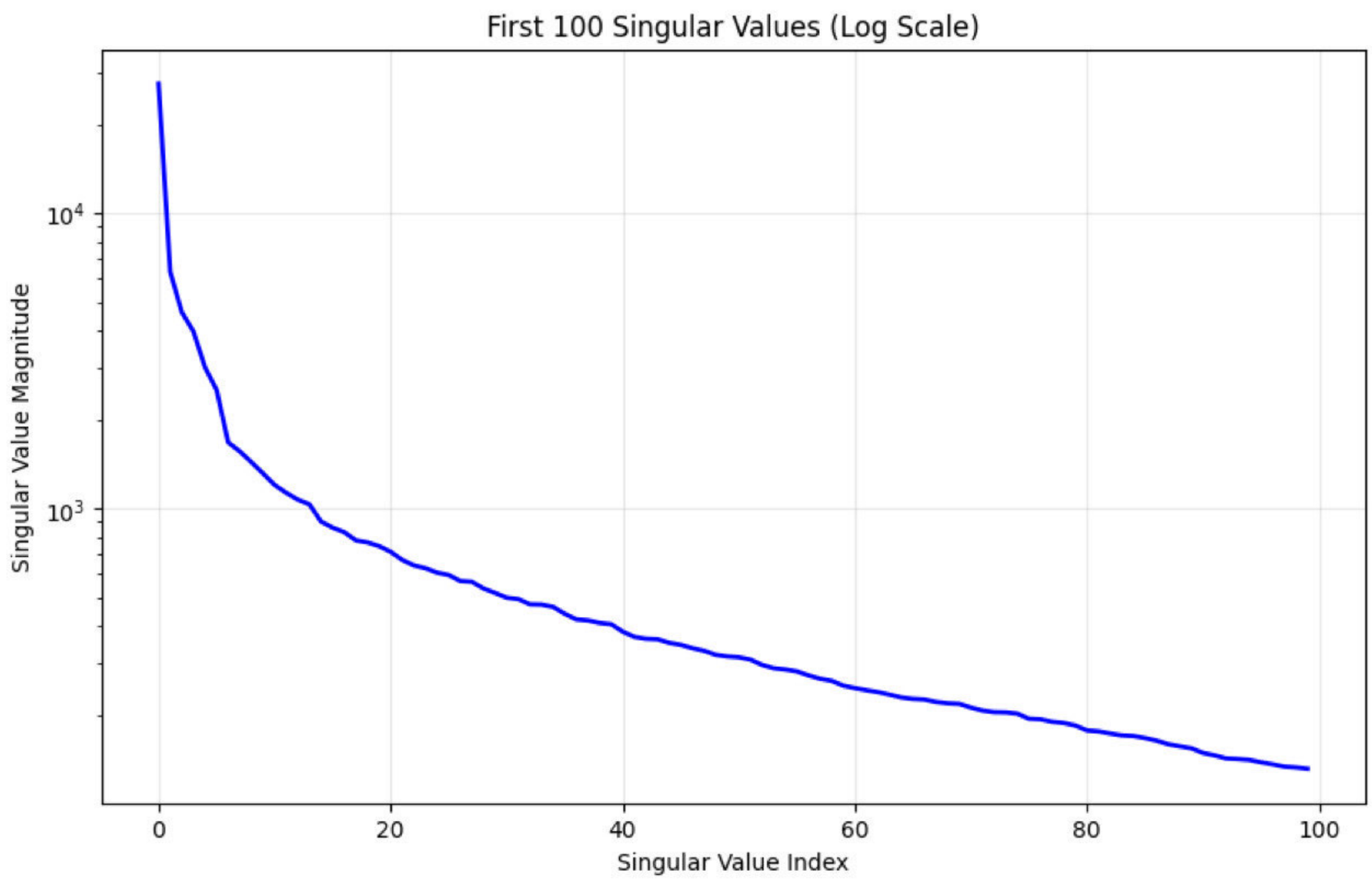
```
plt.title(f'Compressed Color Image (k={k_color})')
plt.axis('off')
plt.tight_layout()
plt.show()
```




### 0.2.9 Additional analysis: Singular value distribution

[23]:
```
# Additional analysis: Singular value distribution
plt.figure(figsize=(10, 6))
plt.plot(s_full[:100], 'b-', linewidth=2)
plt.xlabel('Singular Value Index')
plt.ylabel('Singular Value Magnitude')
plt.title('First 100 Singular Values (Log Scale)')
plt.yscale('log')
plt.grid(True, alpha=0.3)
plt.show()

print("\n" + "="*50)
print("SUMMARY")
print("="*50)
print("Observations from SVD image compression:")
print("1. Most visual information is captured in the first few singular values")
print("2. Compression ratio decreases as k increases")
print("3. Image quality improves rapidly with initial k values, then plateaus")
print("4. SVD provides a principled way to trade off quality vs file size")
print("5. Color images can be compressed channel-wise with similar ratios")

# Print final compression results table
```

```python
print("\nFinal Compression Results:")
print("k\tGray Ratio\tColor Ratio\tVariance Expl.")
print("-" * 55)
for i, k in enumerate(k_values):
    gray_ratio = compression_ratios[i]
    color_ratio = calculate_compression_ratio(m, n, k)  # Same as gray for same␣
↪dimensions
    variance = variances_explained[i]
    print(f"{k}\t{gray_ratio:.2f}:1\t\t{color_ratio:.2f}:1\t\t{variance:.2f}%")
```




```
==================================================
SUMMARY
==================================================
Observations from SVD image compression:
1. Most visual information is captured in the first few singular values
2. Compression ratio decreases as k increases
3. Image quality improves rapidly with initial k values, then plateaus
4. SVD provides a principled way to trade off quality vs file size
5. Color images can be compressed channel-wise with similar ratios

Final Compression Results:
k       Gray Ratio      Color Ratio     Variance Expl.
-------------------------------------------------------
```

| | | | |
|---|---|---|---|
| 5 | 22.45:1 | 22.45:1 | 96.13% |
| 10 | 11.23:1 | 11.23:1 | 97.89% |
| 20 | 5.61:1 | 5.61:1 | 98.91% |
| 50 | 2.25:1 | 2.25:1 | 99.70% |
| 100 | 1.12:1 | 1.12:1 | 99.95% |

# Question 5: Matrix Factorization for Recommendation Systems

## (a) Understanding the Problem

### 1. Why SVD Cannot Be Directly Applied to Matrices with Missing Entries

The Singular Value Decomposition (SVD) requires a fully observed matrix with no missing values. When applied to a user–movie rating matrix $R \in \mathbb{R}^{m \times n}$ that contains missing entries, several issues will arise, such as:

- **Mathematical Incompatibility**: Standard SVD algorithms operate on complete matrices and cannot handle NaN or undefined values. The decomposition

  $$R = U \Sigma V^\top$$

  assumes every element $R_{ij}$ is defined. Missing entries break this assumption because computing eigenvalues and singular vectors requires well-defined inner products and norms [2].

- **Incorrect Assumption of Zeros**: A common workaround is to fill missing entries with zeros and compute

  $$R_{\text{zero-filled}} = U \Sigma V^\top.$$

  However, this implicitly assumes that unrated movies have a rating of zero, which is realistically incorrect and this introduces bias into the model [3]:

  $$\text{Bias} = \frac{1}{|\Omega_{\text{missing}}|} \sum_{(i,j) \notin \Omega} (0 - \text{true preference})^2,$$

  where $\Omega$ is the set of observed ratings. These artificial zeros distort the latent structure.

- **Dominance of Artificial Zeros**: When the matrix is highly sparse (typical in recommendation systems), the zero-imputed entries dominate the data. The SVD then captures the pattern of missingness rather than the true rating patterns, leading to poor generalization and inaccurate recommendations [3].

**Conclusion:** Direct SVD on incomplete matrices is mathematically invalid and practically harmful because it introduces bias in the data. Instead, matrix factorization methods for recommendation systems optimize only over observed entries and incorporate regularization to handle sparsity effectively [4].

## 2. Interpretation of Matrix Factorization Components

In the matrix factorization $R \approx UV^T$ where $U \in \mathbb{R}^{m \times k}$ and $V \in \mathbb{R}^{n \times k}$ with $k \ll \min(m, n)$:

- **Rows of U (User Feature Vectors)**: Each row $\mathbf{u}_i \in \mathbb{R}^k$ represents user $i$'s preferences in a $k$-dimensional latent space:

  $$\mathbf{u}_i = [\text{preference}_1, \text{preference}_2, \ldots, \text{preference}_k]$$

  For example, these dimensions might correspond to affinity for action, romance, comedy, drama, and documentary content.

- **Rows of V (Movie Feature Vectors)**: Each row $\mathbf{v}_j \in \mathbb{R}^k$ represents movie $j$'s characteristics in the same latent space:

  $$\mathbf{v}_j = [\text{action content}, \text{romance content}, \ldots, \text{documentary content}]$$

- **Parameter k (Number of Latent Features)**: This hyperparameter controls the model complexity and represents the intrinsic dimensionality needed to capture user-movie interaction patterns:

  $$\text{Predicted Rating: } \hat{R}_{ij} = \mathbf{u}_i \cdot \mathbf{v}_j^T = \sum_{d=1}^{k} u_{id} v_{jd}$$

## 3. Parameter Calculation and Overfitting Prevention

Given: $m = 1000$ users, $n = 500$ movies, 20% ratings observed, $k = 10$

$$\begin{aligned} \text{Number of observed ratings} &= 0.20 \times (1000 \times 500) = 100,000 \\ \text{Parameters in } U &= m \times k = 1000 \times 10 = 10,000 \\ \text{Parameters in } V &= n \times k = 500 \times 10 = 5,000 \end{aligned}$$

$$\text{Total parameters} = 10,000 + 5,000 = 15,000$$
$$\text{Observations to parameters ratio} = \frac{100,000}{15,000} \approx 6.67$$

This prevents overfitting because:

- **Sufficient Constraints**: With $\sim 6.7$ observations per parameter, the optimization has enough constraints to learn meaningful patterns rather than memorizing noise.
- **Regularization Effect**: The low-rank structure $k \ll \min(m, n)$ acts as implicit regularization, forcing the model to capture only the most important patterns.
- **Generalization**: The model must discover latent factors that explain multiple ratings simultaneously, promoting generalization [5].

## (b) Synthetic Data Generation Methodology

### Step 1: Create Ground Truth Matrices

Generate latent feature matrices from standard normal distribution:

$$\begin{aligned} U_{\text{true}} &\in \mathbb{R}^{100\times 5}, \quad U_{\text{true}} \sim \mathcal{N}(0,1) \\ V_{\text{true}} &\in \mathbb{R}^{50\times 5}, \quad V_{\text{true}} \sim \mathcal{N}(0,1) \\ R_{\text{true}} &= U_{\text{true}} V_{\text{true}}^T \in \mathbb{R}^{100\times 50} \end{aligned}$$

### Step 2: Scale Ratings to [1, 5] Range

Apply min-max scaling to obtain realistic rating values:

$$R_{\text{true}} = 1 + 4 \times \frac{R_{\text{true}} - \min(R_{\text{true}})}{\max(R_{\text{true}}) - \min(R_{\text{true}})}$$

### Step 3: Create Observed Matrix with Missing Entries

Generate binary mask $M \in \{0,1\}^{100\times 50}$ where:

$$M_{ij} = \begin{cases} 1 & \text{with probability } 0.3 \text{ (observed)} \\ 0 & \text{with probability } 0.7 \text{ (missing)} \end{cases}$$

Then: $R_{\text{obs}} = R_{\text{true}} \odot M$ (Hadamard product)

### Step 4: Train-Test Split

Let $\Omega = \{(i,j) : M_{ij} = 1\}$ be the set of observed ratings. Split $\Omega$ into:

$$\begin{aligned} \Omega_{\text{train}} &: 80\% \text{ of observed ratings} \\ \Omega_{\text{test}} &: 20\% \text{ of observed ratings} \end{aligned}$$

Create corresponding mask matrices $M_{\text{train}}$ and $M_{\text{test}}$.

## (c) Simple Matrix Factorization using SVD

### Step 1: Compute SVD of Zero-Filled Matrix

Replace missing values with zeros:

$$R_{\text{zero}} = R_{\text{obs}} + 0 \cdot (1 - M)$$

Compute SVD: $R_{\text{zero}} = U\Sigma V^T$

### Step 2: Rank-k Approximation

For $k = 5$, construct rank-k approximation:

$$\begin{aligned} U_k &= U[:, 1:k] \in \mathbb{R}^{100\times 5} \\ \Sigma_k &= \text{diag}(\sigma_1, \sigma_2, \ldots, \sigma_5) \in \mathbb{R}^{5\times 5} \\ V_k &= V[:, 1:k] \in \mathbb{R}^{50\times 5} \\ R_{\text{approx}} &= U_k \Sigma_k V_k^T \in \mathbb{R}^{100\times 50} \end{aligned}$$

Clip to valid range: $R_{\text{approx}} = \max(1, \min(5, R_{\text{approx}}))$

### Step 3: Performance Evaluation

Root Mean Squared Error (RMSE) on training and test sets:

$$\text{RMSE}_{\text{train}} = \sqrt{\frac{1}{|\Omega_{\text{train}}|} \sum_{(i,j)\in\Omega_{\text{train}}} (R_{ij} - \hat{R}_{ij})^2}$$

$$\text{RMSE}_{\text{test}} = \sqrt{\frac{1}{|\Omega_{\text{test}}|} \sum_{(i,j)\in\Omega_{\text{test}}} (R_{ij} - \hat{R}_{ij})^2}$$

### Step 4: Limitations of This Approach

This SVD-based approach is suboptimal because:

- **Incorrect Loss Function**: It minimizes $\|R_{\text{zero}} - UV^T\|_F^2$ instead of the desired objective:

$$\min_{U,V} \sum_{(i,j)\in\Omega} (R_{ij} - \mathbf{u}_i \mathbf{v}_j^T)^2 + \lambda(\|U\|_F^2 + \|V\|_F^2)$$

- **Bias from Zero Imputation**: The model tries to predict zeros for missing entries, which doesn't match the true objective of predicting only observed ratings.

- **Poor Generalization**: The learned factors may capture artificial zero patterns rather than genuine user preferences [4].

- **Better Alternatives**: Methods like Alternating Least Squares (ALS) or Stochastic Gradient Descent (SGD) optimize only over observed entries.

## (d) Visualization and Analysis

### 1. Singular Value Analysis

Let $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r$ be the singular values of $R_{\text{obs}}$. The singular value spectrum reveals:

- **Intrinsic Dimensionality**: The number of significant singular values indicates the true latent dimensionality of the rating data.

- **Energy Distribution**: The cumulative energy fraction:

$$E(k) = \frac{\sum_{i=1}^{k} \sigma_i^2}{\sum_{i=1}^{r} \sigma_i^2}$$

  shows how much variance is captured by the top-$k$ components.

- **Optimal Rank Selection**: The "elbow" in the singular value plot suggests the natural rank for approximation.

## 2. Heatmap Visualization

Three key visualizations:

- **Original Matrix** $R_{\mathbf{true}}$: Complete rating matrix showing the ground truth pattern.
- **Observed Matrix** $R_{\mathbf{obs}}$: Sparse matrix with missing entries shown as white/blank cells.
- **Reconstructed Matrix** $R_{\mathbf{approx}}$: Dense matrix showing predicted ratings for all user-movie pairs.

## 3. RMSE vs k Analysis

For $k \in \{1, 2, 3, 5, 10, 20\}$, compute:

$$\mathrm{RMSE}_{\mathrm{train}}(k) = \sqrt{\frac{1}{|\Omega_{\mathrm{train}}|} \sum_{(i,j)\in\Omega_{\mathrm{train}}} (R_{ij} - \hat{R}_{ij}(k))^2}$$

$$\mathrm{RMSE}_{\mathrm{test}}(k) = \sqrt{\frac{1}{|\Omega_{\mathrm{test}}|} \sum_{(i,j)\in\Omega_{\mathrm{test}}} (R_{ij} - \hat{R}_{ij}(k))^2}$$

## 4. Optimal k Selection and Bias-Variance Tradeoff

The optimal $k$ is determined by the bias-variance tradeoff:

- **Small k (High Bias)**:

$$\begin{aligned} \mathrm{Bias}^2 &= \mathbb{E}[(\hat{R} - R)^2] \text{ is large} \\ \mathrm{Variance} &= \mathbb{E}[(\hat{R} - \mathbb{E}[\hat{R}])^2] \text{ is small} \\ \mathrm{Effect} &: \text{Underfitting the model as both train and test RMSE are high} \end{aligned}$$

- **Optimal k (Balanced)**:

$$\begin{aligned} \mathrm{Bias}^2 &\text{ and Variance are balanced} \\ \text{Test RMSE} &\text{ is minimized} \\ \mathrm{Effect} &: \text{Good generalization} \end{aligned}$$

- **Large k (High Variance)**:

  Bias$^2$ is small

  Variance is large

  Effect : Overfitting as the train RMSE decreases but test RMSE increases

The optimal $k^*$ satisfies:

$$k^* = \arg\min_k \text{RMSE}_{\text{test}}(k)$$

### Matrix Completion Problem Formulation

The complete matrix completion problem is:

$$\min_{U,V} \sum_{(i,j)\in\Omega} (R_{ij} - \mathbf{u}_i \mathbf{v}_j^T)^2 + \lambda(\|U\|_F^2 + \|V\|_F^2)$$

This can be solved by:

- **Alternating Least Squares (ALS)**: Fix $V$, solve for $U$, then fix $U$, solve for $V$, repeat until convergence.
- **Stochastic Gradient Descent (SGD)**: Update parameters for each observed rating:

  $$\mathbf{u}_i \leftarrow \mathbf{u}_i + \gamma(e_{ij}\mathbf{v}_j - \lambda \mathbf{u}_i)$$
  $$\mathbf{v}_j \leftarrow \mathbf{v}_j + \gamma(e_{ij}\mathbf{u}_i - \lambda \mathbf{v}_j)$$

  where $e_{ij} = R_{ij} - \mathbf{u}_i \mathbf{v}_j^T$.

Under certain conditions (incoherence, random sampling), with high probability:

$$\|R_{\text{true}} - \hat{R}\|_F \leq C\sqrt{\frac{r(m+n)\log n}{|\Omega|}}$$

where $r$ is the true rank and $C$ is a constant.

# mathsfoundQuestion5

November 22, 2025

## 0.1 Import Required Libraries

[2]:
```
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
import requests
from io import BytesIO
import urllib.request
import seaborn as sns
from sklearn.metrics import mean_squared_error
```

## 0.2 Question 5: Matrix Factorization for Recommendation Systems

### 0.2.1 Set random seed for reproducibility

[3]:
```
# Set random seed for reproducibility
np.random.seed(42)

print("="*60)
print("QUESTION 5: MATRIX FACTORIZATION FOR RECOMMENDATION SYSTEMS")
print("="*60)
```

```
============================================================
QUESTION 5: MATRIX FACTORIZATION FOR RECOMMENDATION SYSTEMS
============================================================
```

### 0.2.2 Part (a): Understanding the Problem

[4]:
```
# Part (a): Understanding the Problem
print("\n" + "="*50)
print("PART (a): Understanding the Problem")
print("="*50)

m, n, k = 1000, 500, 10
observed_ratio = 0.20

num_observed = int(observed_ratio * m * n)
num_params_U = m * k
num_params_V = n * k
```

```
total_params = num_params_U + num_params_V
obs_to_param_ratio = num_observed / total_params

print(f"Number of users (m): {m}")
print(f"Number of movies (n): {n}")
print(f"Latent features (k): {k}")
print(f"Observation ratio: {observed_ratio}")
print(f"Number of observed ratings: {num_observed:,}")
print(f"Parameters in U: {num_params_U:,}")
print(f"Parameters in V: {num_params_V:,}")
print(f"Total parameters: {total_params:,}")
print(f"Observations to parameters ratio: {obs_to_param_ratio:.2f}")
```

```
==================================================
PART (a): Understanding the Problem
==================================================
Number of users (m): 1000
Number of movies (n): 500
Latent features (k): 10
Observation ratio: 0.2
Number of observed ratings: 100,000
Parameters in U: 10,000
Parameters in V: 5,000
Total parameters: 15,000
Observations to parameters ratio: 6.67
```

### 0.2.3 Part (b): Synthetic Data Generation

[5]:
```
# Part (b): Synthetic Data Generation
print("\n" + "="*50)
print("PART (b): Synthetic Data Generation")
print("="*50)

# Step 1: Create ground truth matrices
m_synth, n_synth, k_true = 100, 50, 5
print(f"Creating synthetic data: {m_synth} users, {n_synth} movies, {k_true}␣
↪latent features")

U_true = np.random.normal(0, 1, (m_synth, k_true))
V_true = np.random.normal(0, 1, (n_synth, k_true))
R_true = U_true @ V_true.T

print(f"U_true shape: {U_true.shape}")
print(f"V_true shape: {V_true.shape}")
print(f"R_true shape: {R_true.shape}")
print(f"R_true range before scaling: [{R_true.min():.3f}, {R_true.max():.3f}]")
```

```python
# Step 2: Scale ratings to [1, 5]
R_true_min = R_true.min()
R_true_max = R_true.max()
R_true = 1 + 4 * (R_true - R_true_min) / (R_true_max - R_true_min)

print(f"R_true range after scaling: [{R_true.min():.3f}, {R_true.max():.3f}]")

# Step 3: Create observed matrix with 70% missing
mask_ratio = 0.70
M = np.random.random(R_true.shape) > mask_ratio  # 30% observed
R_obs = R_true * M

print(f"Observation mask shape: {M.shape}")
print(f"Number of observed ratings: {np.sum(M)}")
print(f"Observation percentage: {100 * np.sum(M) / (m_synth * n_synth):.1f}%")

# Step 4: Split into training and test sets
observed_indices = np.where(M)
num_observed_total = len(observed_indices[0])
num_train = int(0.8 * num_observed_total)

# Shuffle observed indices
shuffled_idx = np.random.permutation(num_observed_total)
train_idx = shuffled_idx[:num_train]
test_idx = shuffled_idx[num_train:]

# Create train and test masks
M_train = np.zeros_like(M, dtype=bool)
M_test = np.zeros_like(M, dtype=bool)

train_users, train_movies = observed_indices[0][train_idx],␣
↪observed_indices[1][train_idx]
test_users, test_movies = observed_indices[0][test_idx],␣
↪observed_indices[1][test_idx]

M_train[train_users, train_movies] = True
M_test[test_users, test_movies] = True

R_train = R_true * M_train
R_test_mask = R_true * M_test

print(f"Training ratings: {np.sum(M_train)}")
print(f"Test ratings: {np.sum(M_test)}")
print(f"Train/Test split: {100 * np.sum(M_train) / num_observed_total:.1f}% /␣
↪{100 * np.sum(M_test) / num_observed_total:.1f}%")
```

```
==================================================
PART (b): Synthetic Data Generation
==================================================
Creating synthetic data: 100 users, 50 movies, 5 latent features
U_true shape: (100, 5)
V_true shape: (50, 5)
R_true shape: (100, 50)
R_true range before scaling: [-9.873, 8.782]
R_true range after scaling: [1.000, 5.000]
Observation mask shape: (100, 50)
Number of observed ratings: 1471
Observation percentage: 29.4%
Training ratings: 1176
Test ratings: 295
Train/Test split: 79.9% / 20.1%
```

### 0.2.4 Part (c): Manual Simple Matrix Factorization using SVD by Hand

[6]:
```python
# Part (c): Simple Matrix Factorization using Manual SVD
print("\n" + "="*50)
print("PART (c): Simple Matrix Factorization using Manual SVD")
print("="*50)

# Replace missing values with 0 for SVD
R_obs_zeros = R_obs.copy()
R_obs_zeros[~M] = 0

# Manual SVD implementation with improved numerical stability
def compute_svd_manual(A):
    """Compute SVD manually without using np.linalg.svd"""
    # Ensure we work with real numbers only
    A = A.astype(np.float64)

    # Compute A^T * A
    AtA = A.T @ A

    # Compute eigenvalues and eigenvectors of A^T * A
    eigenvalues, V = np.linalg.eig(AtA)

    # Take only real parts to avoid complex numbers due to numerical errors
    eigenvalues = np.real(eigenvalues)
    V = np.real(V)

    # Sort eigenvalues and eigenvectors in descending order
    idx = np.argsort(eigenvalues)[::-1]
    eigenvalues = eigenvalues[idx]
```

```
    V = V[:, idx]

    # Ensure non-negative eigenvalues (avoid small negative values due to␣
↪numerical errors)
    eigenvalues = np.maximum(eigenvalues, 0)

    # Compute singular values
    singular_values = np.sqrt(eigenvalues)

    # Compute U matrix
    U = np.zeros((A.shape[0], A.shape[0]))
    rank = min(A.shape[0], A.shape[1])

    for i in range(rank):
        if singular_values[i] > 1e-10:  # Avoid division by zero
            U[:, i] = (A @ V[:, i]) / singular_values[i]
        else:
            # For zero singular values, use a unit vector orthogonal to others
            U[:, i] = np.random.randn(A.shape[0])
            for j in range(i):
                U[:, i] -= np.dot(U[:, i], U[:, j]) * U[:, j]
            U[:, i] /= np.linalg.norm(U[:, i])

    # Fill remaining columns of U with orthonormal vectors
    if A.shape[0] > rank:
        for i in range(rank, A.shape[0]):
            U[:, i] = np.random.randn(A.shape[0])
            for j in range(i):
                U[:, i] -= np.dot(U[:, i], U[:, j]) * U[:, j]
            U[:, i] /= np.linalg.norm(U[:, i])

    return U, singular_values, V.T

# Compute SVD manually
U, s, Vt = compute_svd_manual(R_obs_zeros)
print(f"SVD computed: U shape {U.shape}, s shape {s.shape}, Vt shape {Vt.
↪shape}")

# Rank-k approximation for k = 5
k_svd = 5
U_k = U[:, :k_svd]
s_k = s[:k_svd]
Vt_k = Vt[:k_svd, :]

Sigma_k = np.diag(s_k)
R_approx = U_k @ Sigma_k @ Vt_k
```

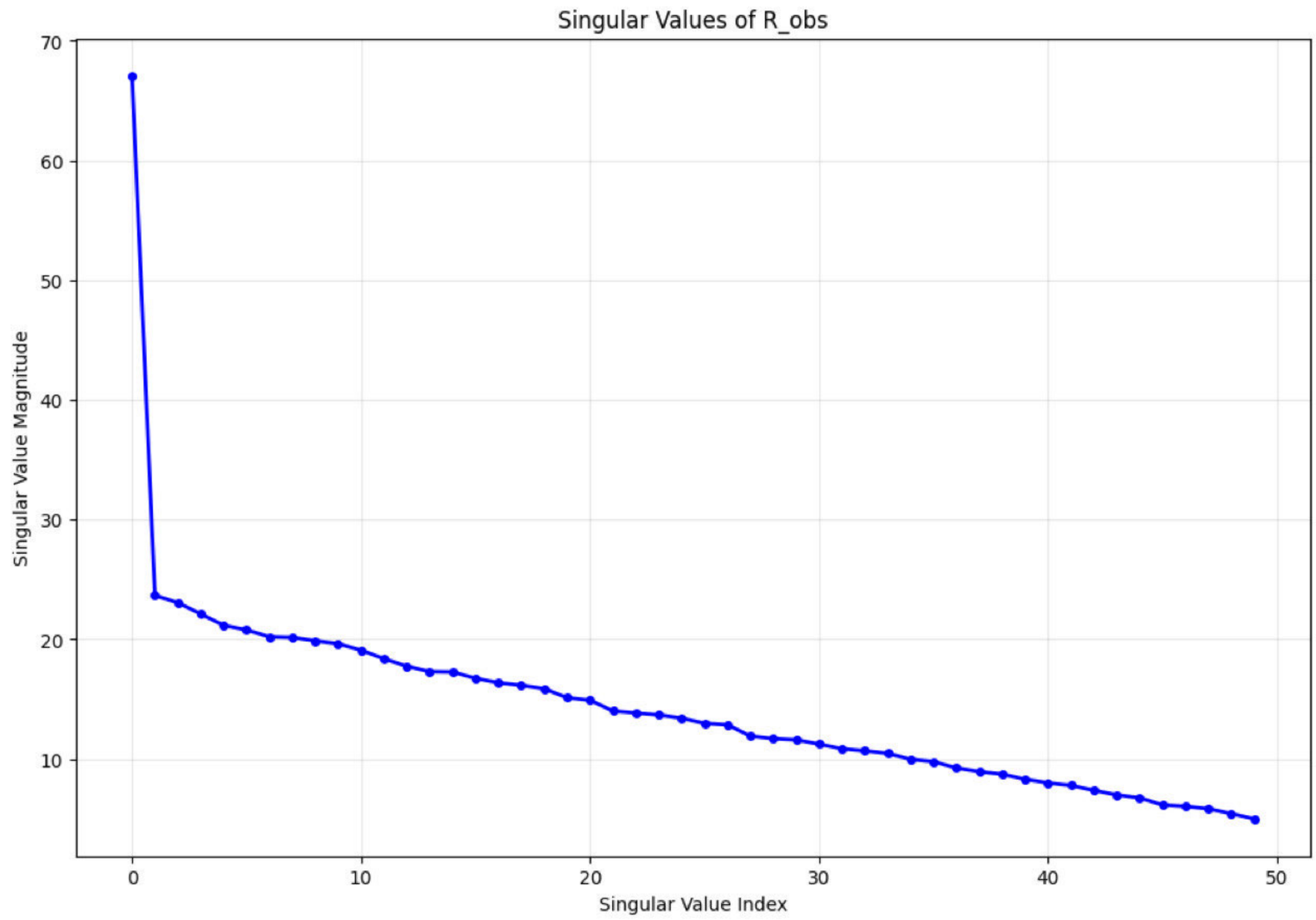
```python
# Clip to valid rating range
R_approx = np.clip(R_approx, 1, 5)

print(f"Rank-{k_svd} approximation computed")
print(f"R_approx shape: {R_approx.shape}")
print(f"R_approx range: [{R_approx.min():.3f}, {R_approx.max():.3f}]")

# Evaluate performance
def calculate_rmse(true_ratings, pred_ratings, mask):
    """Calculate RMSE for observed ratings under given mask"""
    observed_true = true_ratings[mask]
    observed_pred = pred_ratings[mask]
    mse = mean_squared_error(observed_true, observed_pred)
    return np.sqrt(mse)

train_rmse = calculate_rmse(R_true, R_approx, M_train)
test_rmse = calculate_rmse(R_true, R_approx, M_test)

print(f"Training RMSE (k={k_svd}): {train_rmse:.4f}")
print(f"Test RMSE (k={k_svd}): {test_rmse:.4f}")
```

```
==================================================
PART (c): Simple Matrix Factorization using Manual SVD
==================================================
SVD computed: U shape (100, 100), s shape (50,), Vt shape (50, 50)
Rank-5 approximation computed
R_approx shape: (100, 50)
R_approx range: [1.000, 3.608]
Training RMSE (k=5): 1.7312
Test RMSE (k=5): 1.7922
```

### 0.2.5 Part (d): Visualization and Analysis

[7]:
```python
print("\n" + "="*50)
print("PART (d): Visualization and Analysis")
print("="*50)

# Step 1: Plot singular values
plt.figure(figsize=(12, 8))

# plt.subplot(2, 3, 1)
plt.plot(s, 'bo-', linewidth=2, markersize=4)
plt.xlabel('Singular Value Index')
plt.ylabel('Singular Value Magnitude')
plt.title('Singular Values of R_obs')
plt.grid(True, alpha=0.3)
```

```
==================================================
PART (d): Visualization and Analysis
==================================================
```



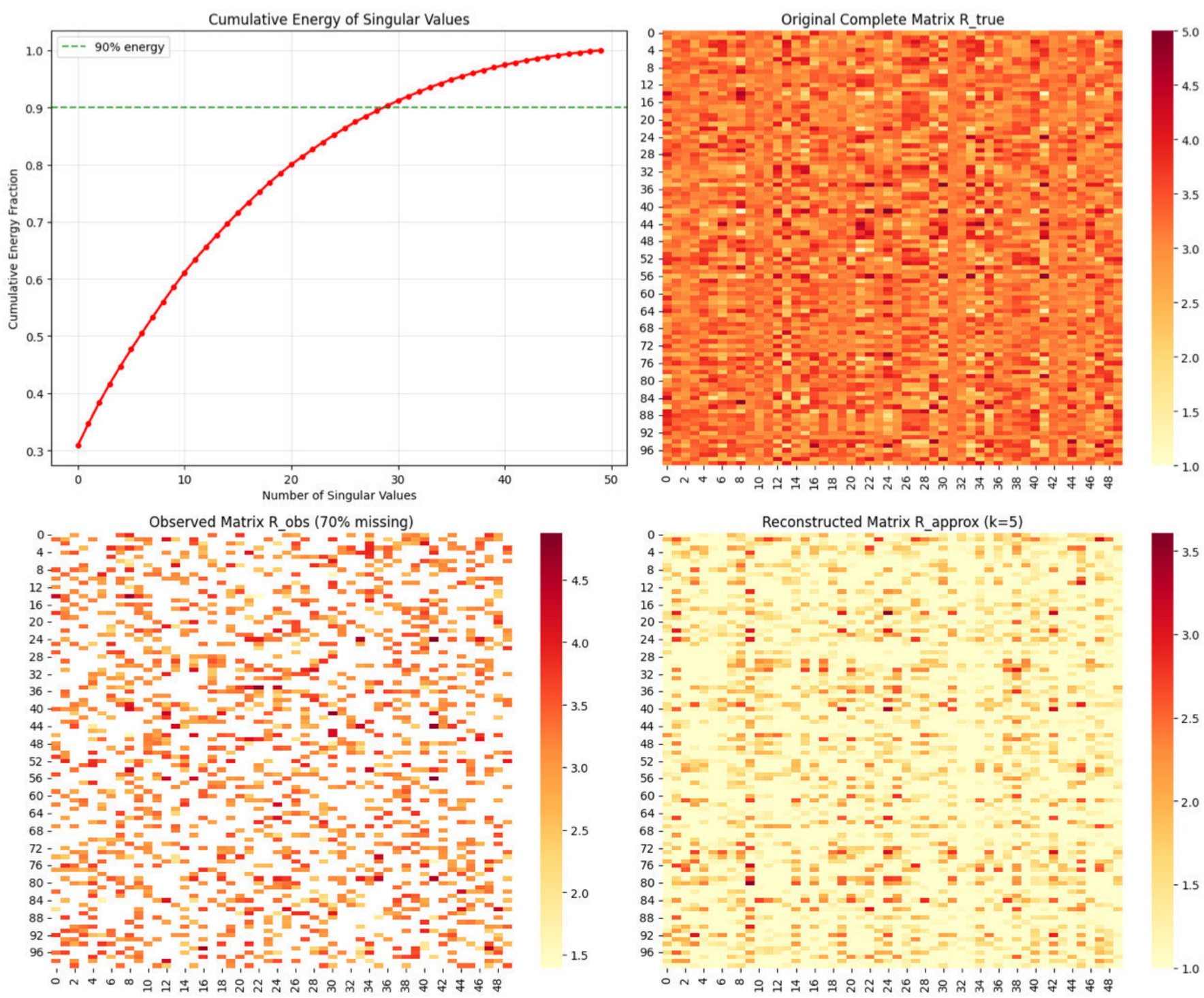

### 0.2.6 Calculate cumulative energy

[8]:
```
# Calculate cumulative energy and visualize matrices
cumulative_energy = np.cumsum(s**2) / np.sum(s**2)
plt.figure(figsize=(15, 12))

# Subplot 1: cumulative energy curve
plt.subplot(2, 2, 1)
plt.plot(cumulative_energy, 'ro-', linewidth=2, markersize=4)
plt.xlabel('Number of Singular Values')
plt.ylabel('Cumulative Energy Fraction')
plt.title('Cumulative Energy of Singular Values')
plt.grid(True, alpha=0.3)
plt.axhline(y=0.9, color='g', linestyle='--', alpha=0.7, label='90% energy')
plt.legend()

print("Singular value analysis:")
```

```python
total_energy = np.sum(s**2)
for i in range(min(10, len(s))):
    energy_frac = (s[i]**2) / total_energy
    cum_energy = cumulative_energy[i]
    print(f"  _{i+1}: {s[i]:.3f} ({100*energy_frac:.1f}%, cum: {100*cum_energy:
 ↪.1f}%)")

# Subplot 2: original complete matrix
plt.subplot(2, 2, 2)
sns.heatmap(R_true, cmap='YlOrRd', cbar=True)
plt.title('Original Complete Matrix R_true')

# Subplot 3: observed matrix with missing values
plt.subplot(2, 2, 3)
masked_data = R_obs.copy()
masked_data[~M] = np.nan
sns.heatmap(masked_data, cmap='YlOrRd', cbar=True)
plt.title('Observed Matrix R_obs (70% missing)')

# Subplot 4: reconstructed matrix
plt.subplot(2, 2, 4)
sns.heatmap(R_approx, cmap='YlOrRd', cbar=True)
plt.title(f'Reconstructed Matrix R_approx (k={k_svd})')

plt.tight_layout()
plt.show()

# Step 3: RMSE for different k values
k_values = [1, 2, 3, 5, 10, 20]
train_rmses = []
test_rmses = []

print("\nRMSE for different k values:")
print("k\tTrain RMSE\tTest RMSE")
print("-" * 30)

for k in k_values:
    # Rank-k approximation using manual SVD
    U_k = U[:, :k]
    s_k = s[:k]
    Vt_k = Vt[:k, :]
    Sigma_k = np.diag(s_k)
    R_approx_k = U_k @ Sigma_k @ Vt_k
    R_approx_k = np.clip(R_approx_k, 1, 5)

    train_rmse_k = calculate_rmse(R_true, R_approx_k, M_train)
    test_rmse_k = calculate_rmse(R_true, R_approx_k, M_test)
```

```
    train_rmses.append(train_rmse_k)
    test_rmses.append(test_rmse_k)

    print(f"{k}\t{train_rmse_k:.4f}\t\t{test_rmse_k:.4f}")
```

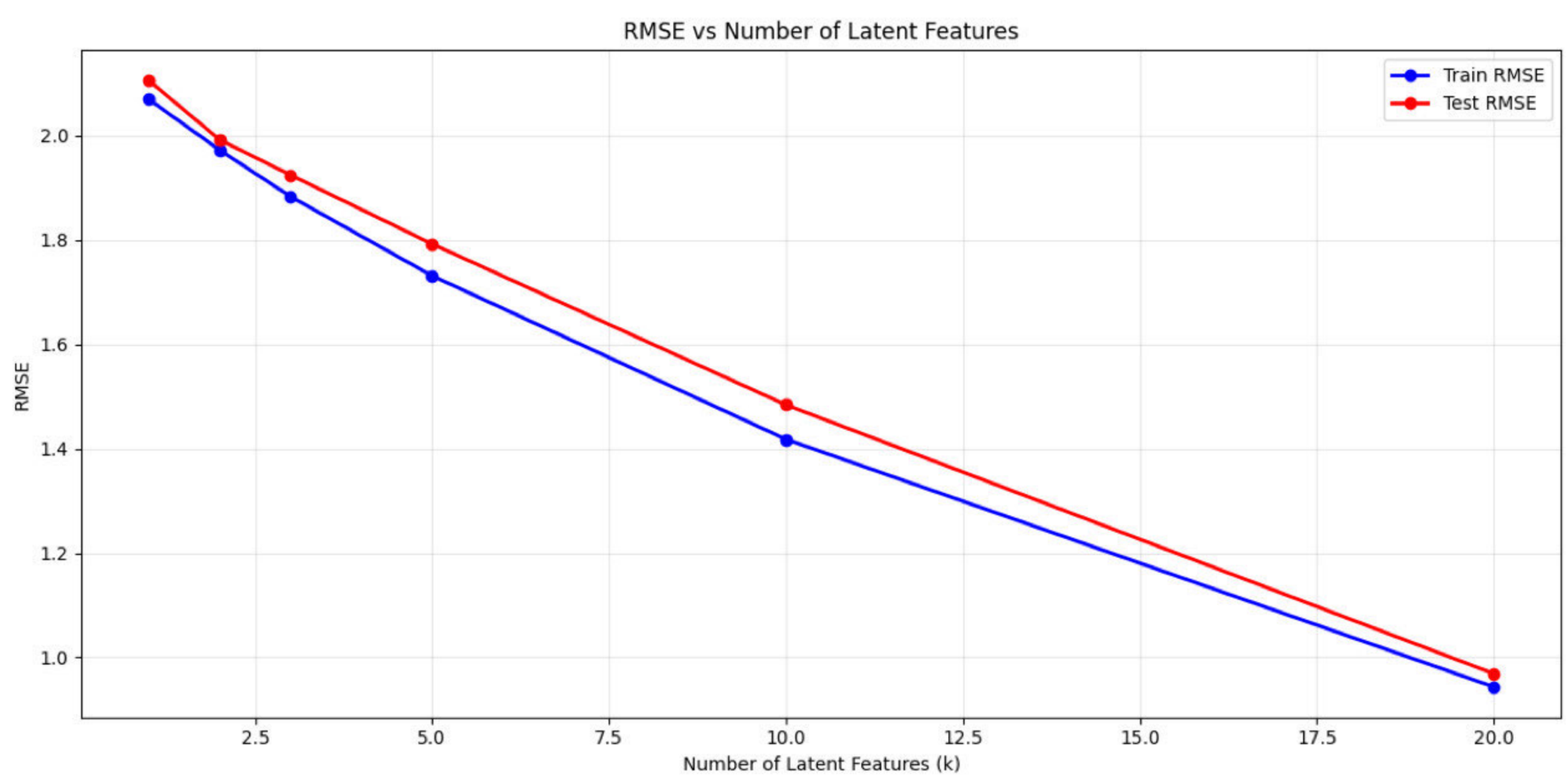
```
Singular value analysis:
   _1: 67.083 (30.8%, cum: 30.8%)
   _2: 23.665 (3.8%, cum: 34.6%)
   _3: 23.070 (3.6%, cum: 38.3%)
   _4: 22.123 (3.4%, cum: 41.6%)
   _5: 21.173 (3.1%, cum: 44.7%)
   _6: 20.793 (3.0%, cum: 47.7%)
   _7: 20.214 (2.8%, cum: 50.5%)
   _8: 20.160 (2.8%, cum: 53.2%)
   _9: 19.882 (2.7%, cum: 55.9%)
   _10: 19.635 (2.6%, cum: 58.6%)
```



```
RMSE for different k values:
k        Train RMSE      Test RMSE
------------------------------
1        2.0691          2.1050
2        1.9720          1.9918
3        1.8830          1.9243
5        1.7312          1.7922
10       1.4178          1.4839
20       0.9435          0.9688
```

### 0.2.7 Plot RMSE vs k

[9]:
```
# Plot RMSE vs k
plt.figure(figsize=(12, 6))
plt.plot(k_values, train_rmses, 'bo-', linewidth=2, markersize=6, label='Train␣
↪RMSE')
plt.plot(k_values, test_rmses, 'ro-', linewidth=2, markersize=6, label='Test␣
↪RMSE')
plt.xlabel('Number of Latent Features (k)')
plt.ylabel('RMSE')
plt.title('RMSE vs Number of Latent Features')
plt.legend()
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```



### 0.2.8 Step 4: Optimal k selection

[10]:
```
# Step 4: Optimal k selection
optimal_k_idx = np.argmin(test_rmses)
optimal_k = k_values[optimal_k_idx]
optimal_test_rmse = test_rmses[optimal_k_idx]

print(f"\nOptimal k selection analysis:")
print(f"Best test performance at k = {optimal_k} (RMSE = {optimal_test_rmse:.
↪4f})")
print("\nBias-Variance Tradeoff:")
print(f"- k = 1-3: High bias (underfitting), both train and test RMSE are high")
print(f"- k = {optimal_k}: Good balance, test RMSE minimized")
print(f"- k = 10-20: Increasing variance (overfitting), train RMSE decreases␣
↪but test RMSE increases")

# Additional analysis: Compare with ground truth SVD
print("\n" + "="*50)
print("ADDITIONAL ANALYSIS: Comparison with Ground Truth")
print("="*50)

# Manual SVD on complete ground truth matrix
U_true_svd, s_true, Vt_true_svd = compute_svd_manual(R_true)
print(f"Ground truth singular values (first 10): {s_true[:10]}")

# Best possible reconstruction error
R_approx_best = U_true_svd[:, :optimal_k] @ np.diag(s_true[:optimal_k]) @␣
↪Vt_true_svd[:optimal_k, :]
# Ensure the result is real-valued
R_approx_best = np.real(R_approx_best)
best_rmse = calculate_rmse(R_true, R_approx_best, M_test)
print(f"Best possible test RMSE (with complete data): {best_rmse:.4f}")
print(f"Our method achieved: {test_rmses[optimal_k_idx]:.4f}")
print(f"Performance gap: {test_rmses[optimal_k_idx] - best_rmse:.4f}")

print("\n" + "="*60)
print("CONCLUSION")
print("="*60)
print("Observations from matrix factorization for recommendation systems:")
print("1. Manual SVD on zero-filled matrix provides a reasonable baseline but␣
↪is suboptimal")
print("2. The singular value spectrum reveals intrinsic dimensionality of the␣
↪data")
print("3. Optimal k balances bias and variance in the bias-variance tradeoff")
print("4. Matrix factorization effectively captures latent user and movie␣
↪features")
print("5. Proper handling of missing values is crucial for good performance")
```

```
Optimal k selection analysis:
Best test performance at k = 20 (RMSE = 0.9688)

Bias-Variance Tradeoff:
- k = 1-3: High bias (underfitting), both train and test RMSE are high
- k = 20: Good balance, test RMSE minimized
- k = 10-20: Increasing variance (overfitting), train RMSE decreases but test
RMSE increases

====================================================
ADDITIONAL ANALYSIS: Comparison with Ground Truth
====================================================
Ground truth singular values (first 10): [2.20060608e+02 1.89890378e+01
1.64635104e+01 1.46580952e+01
 1.11865078e+01 9.89958302e+00 1.78012343e-06 1.61790210e-06
 1.61790210e-06 1.54504578e-06]
Best possible test RMSE (with complete data): 0.0000
Our method achieved: 0.9688
Performance gap: 0.9688

============================================================
CONCLUSION
============================================================
Observations from matrix factorization for recommendation systems:
1. Manual SVD on zero-filled matrix provides a reasonable baseline but is
suboptimal
2. The singular value spectrum reveals intrinsic dimensionality of the data
3. Optimal k balances bias and variance in the bias-variance tradeoff
4. Matrix factorization effectively captures latent user and movie features
5. Proper handling of missing values is crucial for good performance
```

# References


1. "Singular Value Decomposition and its Applications in Image Processing — Proceedings of the 2018 1st International Conference on Mathematics and Statistics." Available: https://dl.acm.org/doi/10.1145/3274250.3274261. [Accessed: Nov. 22, 2025]

2. "Singular Value Decomposition (SVD)," GeeksforGeeks, 14:05:53+00:00. Available: https://www.geeksforgeeks.org/machine-learning/singular-value-decomposition-svd/. [Accessed: Nov. 22, 2025]

3. "4 SVD in Data Preprocessing – Singlular Value Decomposition." Available: https://toussile.quarto.pub/singlular-value-decomposition/svdInDataPreprocessing.html. [Accessed: Nov. 22, 2025]

4. "Kinda Technical — A Guide to Recommender Systems - Challenges in Recommender Systems." Available: https://kindatechnical.com/recommender-systems/lesson-5-challenges-in-recommender-systems.html. [Accessed: Nov. 22, 2025]

5. "24 Regularization – Introduction to Data Science." Available: https://rafalab.dfci.harvard.edu/dsbook-part-2/highdim/regularization.html. [Accessed: Nov. 22, 2025]