

Applied Linear Algebra:

*Application for PCA in Image Compression and Multi Linear Regression in
Data Prediction*



Group 4, Binay Kumar Sah & Kabir Tamari & Arun Kumar Jaiswal
MA017 HT25 Engineering Mathematics
December 5, 2025

Contents

1	Problem 1: Image Compression using PCA	1
1.1	Problem Statement	1
1.2	Background	1
1.3	Methods	2
1.4	Results	2
1.5	Discussion	5
2	Problem 2: Housing price estimation	6
2.1	Problem Statement	6
2.2	Background	6
2.3	Method	6
2.4	Results	7
2.5	Discussion	9
3	Appendix	9
3.1	part1.py	9
3.2	part2.py (Core Logic Only)	12
4	References	13

1 Problem 1: Image Compression using PCA

1.1 Problem Statement

Digital images are collections of pixel intensity values in matrix form. A simple grayscale image can be modeled as a 2D matrix, whereas RGB images are a set of three different 2D matrices. Thus, working on the image dataset requires significant memory and computational resources. Therefore, it is essential to reduce the dimensions of the image without compromising visual information for image data storage, transmission, and analysis.

The primary objective of this problem is to investigate image compression using Principal Component Analysis (PCA), implemented via Singular Value Decomposition (SVD), on an image dataset consisting of 40 sets of grayscale face images, each set comprising 10 facial photos. PCA is applied to the dataset to compute the principal components (also known as eigenfaces). Using only a subset of elements corresponding to the largest eigenvalues, the original images are reconstructed from their compressed representations.

1.2 Background

1.2.1 Principal Component Analysis(PCA)

Principal Component Analysis is a linear algebra method used for dimensionality reduction¹. Its goal is to find a new orthogonal coordinate system in which the variance of the data is maximized along the coordinate axes. Mathematically, if X_c is the centered data matrix, then the covariance matrix of the data is defined as:

$$C = \frac{1}{N} X_c^\top X_c \quad (1)$$

Here, PCA seeks the principal directions known as eigenvectors \mathbf{v}_i and the corresponding covariance captured along each direction known as eigenvalues λ_i

$$C\mathbf{v}_i = \lambda_i \mathbf{v}_i, \quad (2)$$

The principal components with the largest eigenvalues contain most of the important information. Using only these components allows dimensionality reduction while preserving image quality.

1.2.2 Singular Value Decomposition(SVD)

Singular Value Decomposition (SVD)² is a matrix factorization technique that decomposes any matrix \mathbf{X}_c as :

$$X_c = U \Sigma V^\top \quad (3)$$

where,

- $U \in \mathbb{R}^{N \times N}$: Matrix containing the left singular vectors,
- $\Sigma \in \mathbb{R}^{N \times d}$: Diagonal matrix of singular values σ_i .
- $V \in \mathbb{R}^{d \times d}$: Matrix containing the right singular vectors

1.2.3 Relation Between PCA and SVD

From the above relations, the covariance matrix can be written using SVD as:

$$X^T X = V \Sigma^2 V^T \quad (4)$$

and the relationship between eigenvalues and singular values is:

$$\lambda_i = \frac{\sigma_i^2}{N} \quad (5)$$

The columns of V give the principal components. Thus, we can generate an image using only k principal components,

$$X_c \approx U_k \Sigma_k V_k^T, \quad k \ll d. \quad (6)$$

Also, the projection of an image X onto the reduced subspace is:

$$\mathbf{z} = V_k^T (\mathbf{x} - \boldsymbol{\mu}) \quad (7)$$

where, $z \in \mathbb{R}^k$ is the compressed representation

1.3 Methods

A set of images is loaded in the initial stage. For visualization and data analysis, Jupyter Lab is used. After loading the images, PCA and SVD operations are performed, followed by the reconstruction of the images using an approximate value of k

Listing 1: Pseudocode for Image Compression using PCA

```

# Step 1: Load and preprocess data
Load all grayscale images into matrix X (N x d)
Compute mean_face = mean(X, axis=0)
Center data: X_centered = X - mean_face # removes bias, focuses on variance

# Step 2: Perform SVD
U, Sigma, V_T = svd(X_c) # X_c = U * Sigma * V_T
# U: left singular vectors, Sigma: singular values, V_T: right singular vectors (PCs)

# Step 3: Visualize eigenfaces
Plot the Vt Image : plt.imshow(Vt[i].reshape(h, w), cmap='gray')

# Step 4: Select top-k principal components
Compute eigenvalues: lambda_i = (Sigma_i)^2 / N
Sort eigenvalues in descending order
Choose k components corresponding to the largest eigenvalues

# Step 5: Project and reconstruct images
V_k = V_T[:k, :].T # top k principal components
Z = X_c @ V_k      # projection of data onto PCs
X_hat = Z @ V_k.T + mean_image # reconstructed images

# Step 5: Evaluate
Compare the output images with different values of k
Evaluate the performance using Mean Squared Error and Compression ratio.

```

1.4 Results

We perform the PCA operation using SVD on the image set and obtain the top principal components, also known as eigenfaces. These images represent the directions of maximum variance in the dataset and capture the most significant facial features across all faces. Fig. 1

shows the first nine eigenfaces, which highlight common patterns such as the overall face shape, eyes, and mouth regions.

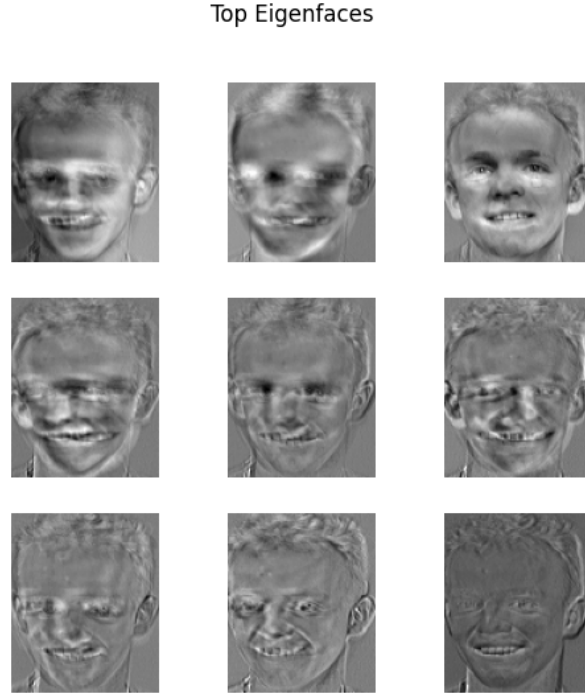


Figure 1: Top principal components (Eigenfaces) visualization

Later, we compressed the image with a different number of principal components k , and the output is shown in Fig. 2. From the results, we can observe that for small values of k , the reconstructed images exhibit a loss of fine details and blurring. However, for moderate values of k such that $k > 40$, the reconstructed faces are visually indistinguishable from the original images.

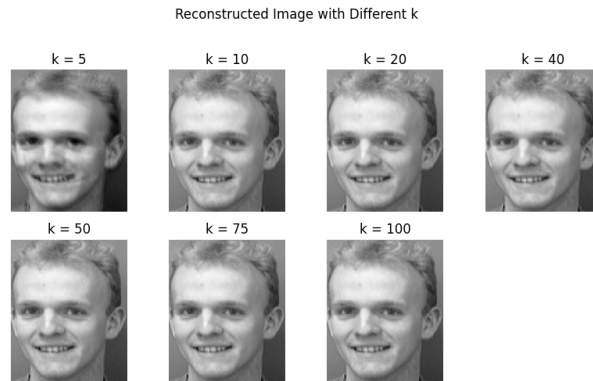


Figure 2: Reconstructed Image with different k

Additionally, we evaluate the performance of the PCA-based image compression method using Mean Squared Error (MSE) and compression ratio.

Mean Squared Error (MSE)

MSE measures the amount of information lost during reconstruction. Mathematically, it can be stated as:

$$\text{MSE} = \frac{1}{Nd} \|X - \hat{X}_k\|_F^2 \quad (8)$$

where,

- X : Original image matrix
- \hat{X}_k : Reconstructed image using k components
- $\|\cdot\|_F$: Frobenius norm

The result as shown in Fig. 3 shows a sharp decrease in MSE when increasing k from a very small value to a moderate one. For $k \geq 10$ the reconstruction error becomes effectively zero, indicating almost perfect reconstruction of the original images. This behavior suggests that only a small number of principal components are sufficient to capture nearly all relevant image information.

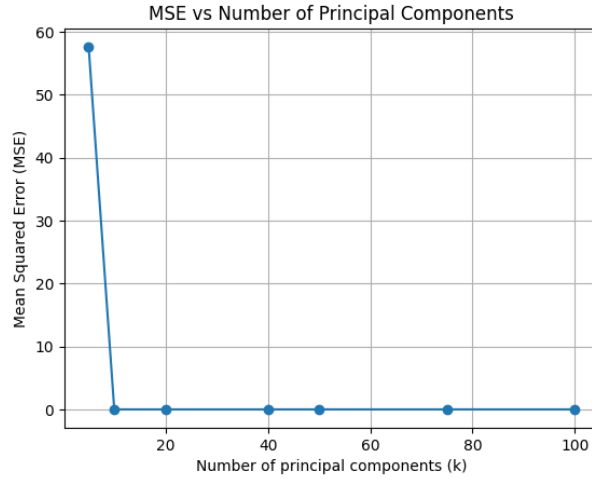


Figure 3: MSE VS Number of Principal Components

Compression Ratio (Storage Efficiency)

It measures how much the data size is reduced by compression, i.e., the efficiency of the compression method in reducing storage or memory requirements. Mathematically,

$$\text{Compression Ratio} = \frac{\text{Original storage}}{\text{Compressed storage}} \quad (9)$$

For a grayscale image $N * d$:

- Original: $N \cdot d$
- Compressed: $k(N + d + 1)$

Fig. 4 shows that the compression ratio decreases monotonically with increasing k . For small values of k , a significant reduction in storage requirements is achieved. However, as more principal components are retained, the compressed representation approaches or even exceeds the size of the original data, reducing the benefit of compression.

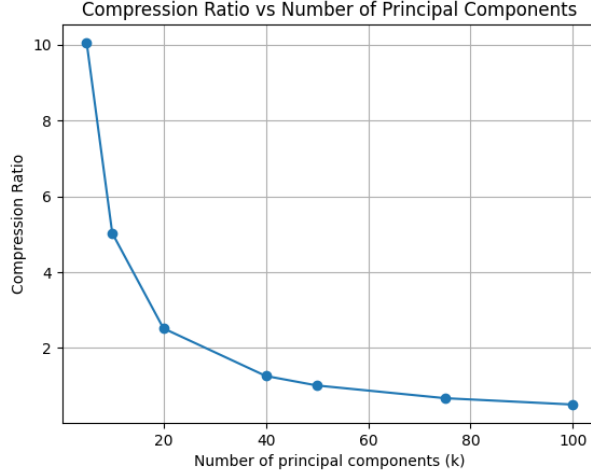


Figure 4: Compression Ratio vs Number of Principal Components

Thus, MSE and compression ratio results together illustrate a clear relation between reconstruction accuracy and storage efficiency. While very small values of k lead to noticeable reconstruction error but high compression, moderate values of k provide near-perfect reconstruction with still reasonable compression ratios. Increasing k beyond this point does not improve reconstruction quality but significantly reduces compression efficiency.

1.5 Discussion

Using the PCA technique with the help of SVD, we were able to reduce the image effectively without compromising important information. However, there are some limitations with this approach.

SVD compression quality strongly depends on the dataset; as a result, if the data images differ significantly from each other, then the image may not be reconstructed in good quality. Additionally, PCA is a linear technique, which means it cannot efficiently capture nonlinear patterns such as edges and textures.

We can improve the reconstruction approach using kernel PCA to capture nonlinear structures. Similarly, applying PCA locally on image patches or combining PCA with other techniques such as wavelet-based methods or deep learning-based autoencoders can help to effectively overcome the limitations of PCA.

2 Problem 2: Housing price estimation

2.1 Problem Statement

We were given a data set using which we built a model to estimate housing prices based on attributes like number of rooms, crime rate, and others that are in the dataset. Specifically, we want to find a linear relationship between the attributes x_n and the prices of the house (y).

$$y = a_0 + a_1x_1 + a_2x_2 + \dots + a_Nx_N$$

Here, y represents the predicted price (in thousands of dollars), x_n are the various features of the house, and a_n are the weights we need to calculate. The goal is to implement this using the Normal Equation method and evaluate how well it works on unseen data. We are also adding a bias a_0 as realistically, a house with "zero" for all features (zero rooms, zero crime, etc.) would be an outlier and will make the model worse if we try to adjust the whole model to fit the outlier data too.

2.2 Background

To solve this, we'll be using the method of least squares. The idea is we want to draw a line or hyperplane through our data points such that the total squared distance between the actual prices and the estimates are as low as possible. The distance between them is what we call an error or residual.

Linear Regression & Bias: We model the price y as a weighted sum of features plus a noise term:

$$y = \mathbf{x}^T \beta + \epsilon$$

Derivation of the Normal Equation: We start with the cost function $E(\beta)$, which represents the sum of squared errors between our predictions ($X\beta$) and the actual values (y):

$$E(\beta) = \|X\beta - y\|^2 = (X\beta - y)^T(X\beta - y)$$

To find the best weights β , we need to minimize this cost. We do this by expanding the equation then taking the gradient of $E(\beta)$ with respect to β we get:

$$\nabla_{\beta} E(\beta) = \frac{\partial}{\partial \beta} (\beta^T X^T X \beta - 2\beta^T X^T y)$$

Setting the gradient to zero to find the value's condition for optimality and finally, we multiply by the inverse of $(X^T X)$ to isolate β we get

$$\beta = (X^T X)^{-1} X^T y$$

This equation gives us the exact mathematical solution for the best-fitting line which is called the Normal Equation. This Normal Equation gives us the best possible solution in a single step, without needing to guess or iterate.

2.3 Method

We load the dataset and split it into two parts, 80% for training the model and 20% for testing it. This ensures we aren't just memorizing the data but actually learning patterns that generalize.

We repeated this process 100 times, shuffling the data each time as the split is random, and single split could give misleading results. Averaging over 100 runs gives us a much more reliable picture of the model's performance.

Listing 2: Pseudocode for Housing Price Estimation

```
# 1. Load Data
data = load_dataset("Problem2_Dataset.csv")
X = data_features
y = data_target

# 2. Split Data (Training/Testing)
X_train, y_train = subset(X, y, 80%)
X_test, y_test = subset(X, y, 20%)

# 3. Add Bias Term (Intercept)
# Append a column of 1s to X
X_train = augment_with_ones(X_train)
X_test = augment_with_ones(X_test)

# 4. Use Normal Equation
# Direct formula for beta
beta = inverse(X_train.T @ X_train) @ (X_train.T @ y_train)

# 5. Predict and Evaluate
y_pred = X_test @ beta
rmse = sqrt(mean((y_pred - y_test)^2))
```

2.4 Results

For the analysis of the model, we used 20% of the data that was separated before the calculation. After running our simulation 100 times, we gathered the following statistics. The target values in this dataset are in \$1000s, so the errors should be interpreted in that context. Using the Root Mean Squared Error (RMSE) as the evaluation metrics for our analysis.

$$\text{RMSE} = \sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2}$$

- **Mean RMSE:** ≈ 5.04

On average, our model's predictions are off by about \$5,040.

- **RMSE Std Dev:** ≈ 0.69

The standard deviation is fairly low (\$690) which means our model is consistent.

- **Best RMSE:** ≈ 3.63

In the best-case scenario (a particularly "easy" split of the data), the error dropped to around \$3,630.

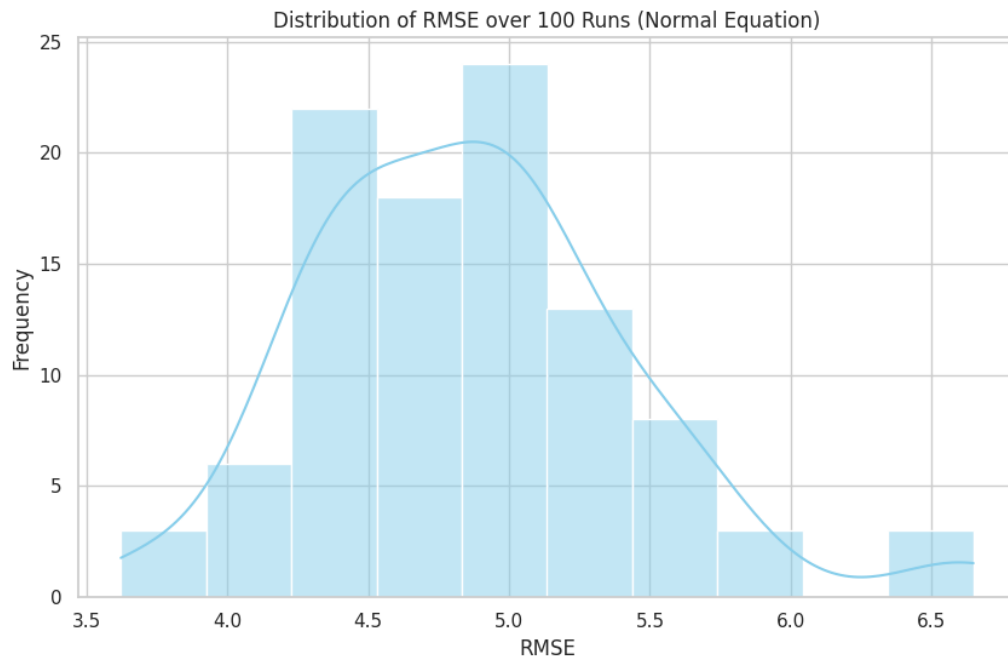


Figure 5: RMSE Distribution over 100 runs. You can see the bell curve shape centered around 5.0, confirming the stability we observed in the statistics.

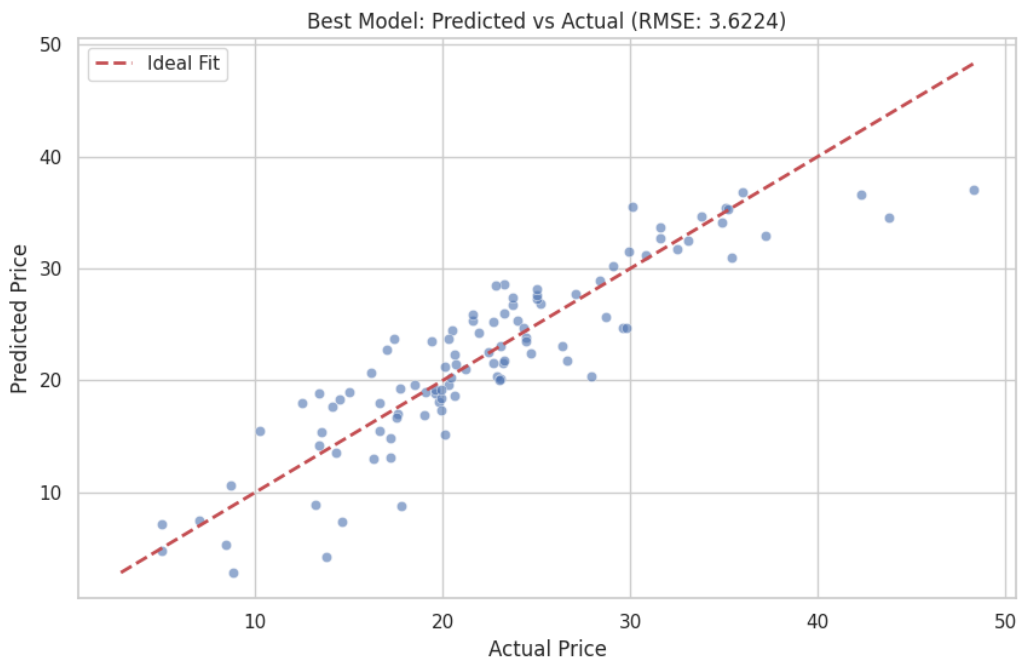


Figure 6: Best Model Predictions. Ideally, all dots would land on the red dashed line. Our model does well for the most part, but it struggles with the highest-priced homes (top right). This suggests there might be some non-linear factors at play for luxury properties.

2.5 Discussion

The normal equation is an effective tool to solve the multi-linear equation but it's computationally heavy for large datasets as inverting a matrix becomes incredibly slow as the number of features grows. Secondly, it handles outliers very poorly, since we are minimizing the squared error a single massive outlier can pull the regression line messing up prediction for everyone else. For larger dataset we switch to gradient decent, since it's an iterative methods and is computationally lighter than normal equation it scale much better.

3 Appendix

3.1 part1.py

```
import numpy as np
import matplotlib.pyplot as plt
import os
from PIL import Image

imageSet = 's5'

# Load all the images of specific set
def load_images(path):
    images = []
    for file in os.listdir(path):
        img = Image.open(os.path.join(path, file)).convert('L') #L -Luminance(black and white)
        images.append(np.array(img)) # Converting images to 2d array
    return np.array(images)

faces = load_images('./Data/Dataset/att_faces/'+imageSet)
print("Dataset shape:", faces.shape)
print(f'face = {faces}')

plt.figure(figsize=(6,6))

for i in range(9):
    plt.subplot(3,3,i+1)
    plt.imshow(faces[i], cmap='gray')
    plt.axis('off')
plt.suptitle('Sample Original Faces')
plt.show()

# Converting images into vectors
# X - row : Each data/image,
# X - col : different pixel values of subject in different data
n_samples, h, w = faces.shape
X = faces.reshape(n_samples, h * w)
print("Data matrix shape:", X.shape)
```

```

# PCA - Mean centering
mean_face = np.mean(X, axis = 0) # axis = 0 -> operate coloumn-wise : take the mean
    of each pixel across all images
X_centered = X - mean_face          # Determine deviance of each pixel value from the
    mean (Eg: 133 - 129.3 = 3.7)
# print(f'Mean face = {mean_face}')
# print(f'X = {X}')
# print(f'X_centered = {X_centered}')

# Applying SVD
    # Vt = eigenfaces
    # S = singular values
    # Eigenvalues = S^2/(n-1)
U, S, Vt = np.linalg.svd(X_centered, full_matrices=False)

# print(f'U = {U}')
# print(f'S = {S}')
# print(f'Vt = {Vt}')

# Visualize eigenfaces
plt.figure(figsize=(6,6))
for i in range(9):
    plt.subplot(3,3,i+1)
    plt.imshow(Vt[i].reshape(h, w), cmap='gray')
    plt.axis('off')
plt.suptitle("Top Eigenfaces")
plt.show()

# Reconstruction function the face

def reconstruct_image(x, mean, Vt, k):
    return mean + x @ Vt[:k].T @ Vt[:k]

# Comparing the different k values
k_values = [5, 10, 20, 40, 50, 75, 100]

img_index = 0
plt.figure(figsize=(10,8))

for i, k in enumerate(k_values):
    recon = reconstruct_image(X_centered[img_index], mean_face, Vt, k)
    plt.subplot(3,4,i+1)
    plt.imshow(recon.reshape(h,w), cmap='gray')
    plt.title(f"k = {k}")
    plt.axis('off')

plt.suptitle("Reconstructed Image with Different k")
plt.show()

#----- Analysis -----#
# Function to compute MSE and Compression ratio

```

```

data = faces.shape
N = data[1]
d = data[2]

def compute_mse(original, reconstruct_image):
    return np.mean((original - reconstruct_image)**2)

def compute_compression_ratio(k):
    original_size = N*d
    compressed_size = k*(N+d+1)
    return original_size / compressed_size

def pca_reconstruct(U, S, Vt, mean_image, k):
    """
    Reconstructs image using first k PCA components.
    """
    Uk = U[:, :k]
    Sk = np.diag(S[:k])
    Vk = Vt[:k, :]

    X_reconstructed = Uk @ Sk @ Vk + mean_image
    return X_reconstructed

# Compute metric for different k
mse_values = []
compression_ratios = []
X_Original = X

for k in k_values:
    recon = pca_reconstruct(U, S, Vt, mean_face, k)
    mse = compute_mse(X_Original, recon)
    cr = compute_compression_ratio(k)

    mse_values.append(mse)
    compression_ratios.append(cr)

print(f'Mse Values are: {mse_values}')
print(f'Compress ratios = {compression_ratios}')

# MSE Vs Number of Components k

plt.figure()
plt.plot(k_values, mse_values, marker='o')
plt.xlabel("Number of principal components (k)")
plt.ylabel("Mean Squared Error (MSE)")
plt.title("MSE vs Number of Principal Components")
plt.grid(True)
plt.show()

# Compression Ratio vs Number of Components
plt.figure()
plt.plot(k_values, compression_ratios, marker='o')

```

```
plt.xlabel("Number of principal components (k)")
plt.ylabel("Compression Ratio")
plt.title("Compression Ratio vs Number of Principal Components")
plt.grid(True)
plt.show()
```

3.2 part2.py (Core Logic Only)

```
import pandas as pd
import numpy as np
import os

FEATURES = [
    "CRIM",
    "ZN",
    "INDUS",
    "CHAS",
    "NOX",
    "RM",
    "AGE",
    "DIS",
    "RAD",
    "TAX",
    "PTRATIO",
    "LSTAT",
]

def load_data(filepath="Problem2_Dataset.csv"):
    if not os.path.exists(filepath):
        print(f"Error: {filepath} not found.")
        return None
    return pd.read_csv(filepath)

def run_once(data):
    x = data[FEATURES].to_numpy()
    y = data["target"].to_numpy()

    indices = np.random.permutation(len(data))
    split = int(0.8 * len(data))

    train_x = x[indices[:split]]
    test_x = x[indices[split:]]
    train_y = y[indices[:split]]
    test_y = y[indices[split:]]

    # Add intercept/bais term
    train_x = np.hstack([np.ones((train_x.shape[0], 1)), train_x])
    test_x = np.hstack([np.ones((test_x.shape[0], 1)), test_x])
    # Normal equation
    # beta = (X^T X)^-1 X^T y
```

```

beta = np.linalg.inv(train_x.T @ train_x) @ (train_x.T @ train_y)
# Predictions
y_pred = test_x @ beta
# Errors
errors = y_pred - test_y
abs_errors = np.abs(errors)
rmse = np.sqrt(np.mean(errors**2))

return beta, abs_errors, rmse, y_pred, test_y

if __name__ == "__main__":
    main_data = load_data()
    if main_data is not None:
        beta, abs_errors, rmse, y_pred, test_y = run_once(main_data)

        print("-" * 50)
        print("Single Run Result (Normal Equation):")
        print("Coefficients (Beta):", beta)
        print("RMSE:", rmse)
        print("Mean Absolute Error:", np.mean(abs_errors))

```

4 References

References

- [1] P. Rizwan Ahmed Khan, *Pca and svd: Explained simply*, <http://www.youtube.com/watch?v=EiYzthtn0dU>, YouTube video. Published: September 15, 2025, 2025.
- [2] V. Kernel, *Svd visualized, singular value decomposition explained / see matrix , chapter 3 some2*, <https://www.youtube.com/watch?v=vSczTbgc8Rc>, YouTube video. Published: April 23, 2022, 2022.