GRAM SCHMIDT ORTHOGONALIZATION, SVD, LEAST SQUARE APPROXIMATION

Mathematical Foundations for Data Science (AZ5402)

ASSIGNMENT

Mirsha A. K. - 2022510044

B.TECH AI & DS

Gram-Schmidt Orthogonalization Based Feature Selection

Introduction Specificities :

Gram-Schmidt Orthogonalization (GSO) is a mathematical technique used in feature selection to create a new set of features that are uncorrelated (orthogonal) to each other while capturing most of the information from the original features. This helps improve machine learning model performance and reduces computational cost by dealing with a smaller, more informative feature set.

The Basic Concept Overview :

Imagine you have a bunch of arrows (features) pointing in different directions. Feature selection with GSO aims to find a new set of arrows that represent the same space but are perpendicular (orthogonal) to each other. These new, uncorrelated features will capture the essential information from the originals without redundancy.

The Process Flow :

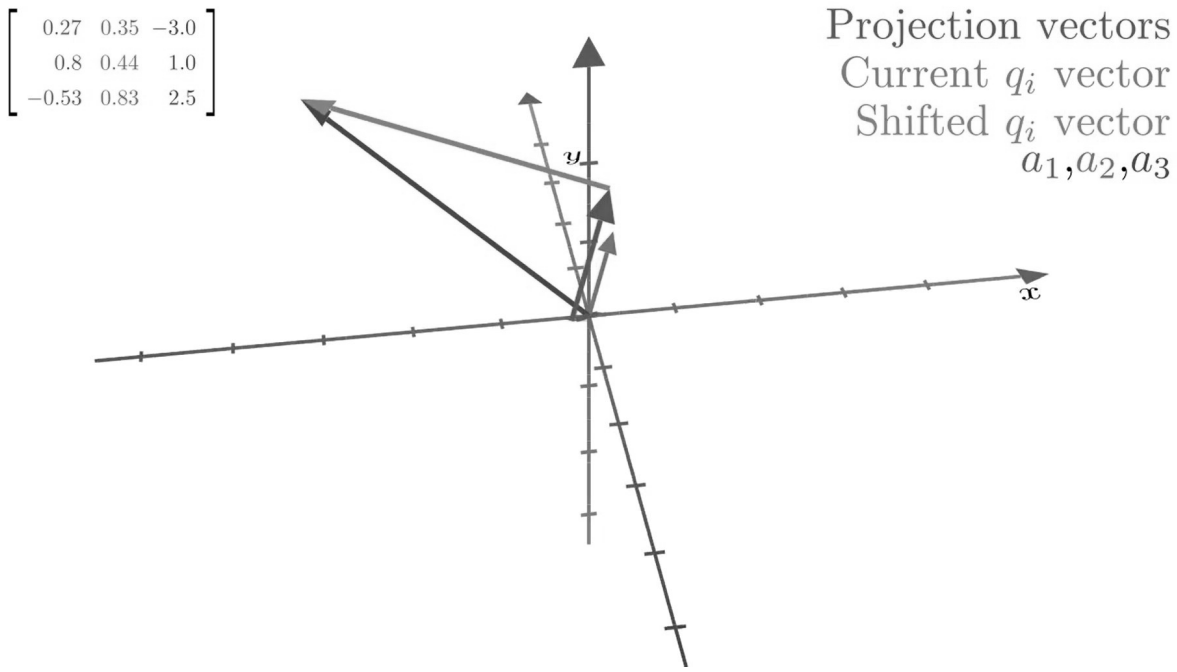
Here's the process:

1. Start with the first original feature vector.

2. Project all subsequent features onto the space spanned by the already chosen features and remove that projection. This ensures the new feature is independent of the previous ones.
3. Normalize the remaining vector to get a unit-length new feature.
4. Repeat steps 2 and 3 for all remaining features.

By the end, you have a set of orthogonal features that effectively capture the important information from the originals.

Mathematical Overview :



Let's represent your original features as a set of vectors `x_1, x_2, ..., x_n` in an n-dimensional space. Here are the relevant formulas involved in GSO for feature selection:

1. **Projection:** The projection of vector `x_i` onto the space spanned by vectors `x_1, ..., x_(j-1)` is:

$$proj_{(x_1, ..., x_{(j-1)})}(x_i) = ((x_i \cdot x_1) / ||x_1||^2) * x_1 + ... + ((x_i \cdot x_{(j-1)}) / ||x_{(j-1)}||^2) * x_{(j-1)}$$

where `.` denotes the dot product and `||x||` represents the vector's magnitude.

2. Removing the projection: To get the part of `x_i` orthogonal to the existing features:

$$x_i' = x_i - proj_{(x_1, ..., x_{(j-1)})}(x_i)$$

3. Normalization (optional): Normalize `x_i'` to get a unit-length vector (useful for some algorithms):

$$x_j = x_i' / ||x_i'||$$

Here, `x_j` represents the new, j-th orthogonal feature vector.

By iteratively applying these equations, you obtain a set of orthogonal feature vectors that can be used in your machine learning model.

Breast Cancer Wisconsin (Diagnostic) Data Set

About Dataset :

The UCI Breast Cancer Wisconsin (Diagnostic) Dataset is a well-known dataset used for machine learning and statistical modeling, particularly for binary classification tasks.

Overview :

Dataset Name: Breast Cancer Wisconsin (Diagnostic)
Source: UCI Machine Learning Repository
Number of Instances: 569
Number of Attributes: 30 numerical features (plus 1 target variable)
Task: Classification (Benign vs. Malignant)

Attributes Information :

The dataset consists of features computed from a digitized image of a fine needle aspirate (FNA) of a breast mass. They describe characteristics of the cell nuclei present in the image. The features are divided into three main categories: mean, standard error, and worst (largest) value. For each cell nucleus, the following features are provided:

- radius (mean of distances from center to points on the perimeter)
- texture (standard deviation of gray-scale values)

- perimeter
- area
- smoothness (local variation in radius lengths)
- compactness (perimeter^2 / area - 1.0)
- concavity (severity of concave portions of the contour)
- concave points (number of concave portions of the contour)
- symmetry
- fractal dimension ("coastline approximation" - 1)

Pythonic Code :

Step 1: Load and preprocess the dataset

First, we'll load the Breast Cancer dataset and select 15 features.

```
import numpy as np
import pandas as pd
from sklearn.datasets import load_breast_cancer
from sklearn.datasets import load_iris
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.ensemble import AdaBoostClassifier

# Load the Breast Cancer dataset
data = load_breast_cancer()
```

```
X = data.data
y = data.target

# Select 15 features
selected_features = data.feature_names[:15]
X_selected = X[:, :15]

# Normalize the features
scaler = StandardScaler()
X_normalized = scaler.fit_transform(X_selected)

# Convert to DataFrame for better visualization
df = pd.DataFrame(X_normalized, columns=selected_features)
df['target'] = y

print(df.head())
```

Step 2: Perform Gram-Schmidt Orthogonalization

Next, we implement the Gram-Schmidt process to orthogonalize the feature vectors.

```
def gram_schmidt(X):
    Q = np.zeros_like(X)
    for i in range(X.shape[1]):
        qi = X[:, i]
        for j in range(i):
            qj = Q[:, j]
            qi -= np.dot(qi, qj) * qj
```

```
        qi /= np.linalg.norm(qi)
        Q[:, i] = qi
    return Q

# Apply Gram-Schmidt process
Q = gram_schmidt(X_normalized)

# Convert to DataFrame for visualization
df_orthogonal = pd.DataFrame(Q, columns=selected_features)
df_orthogonal['target'] = y

print(df_orthogonal.head())
```

Step 3: Measure Orthogonality

We measure the orthogonality of the transformed features.

```
def measure_orthogonality(Q):
    orthogonality_matrix = np.dot(Q.T, Q)
    off_diagonal_elements = orthogonality_matrix -
np.diag(np.diagonal(orthogonality_matrix))
    orthogonality_score = np.sum(np.abs(off_diagonal_elements))
    return orthogonality_score

# Measure orthogonality
orthogonality_score = measure_orthogonality(Q)
print(f'Orthogonality Score: {orthogonality_score}')
```

Output :

```
Orthogonality Score: 5.797767051608088e-14
```

Step 4: Evaluate the Selected Features

We use a classifier to evaluate the performance of the selected features.

```
# Split the dataset
X_train, X_test, y_train, y_test = train_test_split(Q, y, test_size=0.1,
random_state=42)

# Train a Logistic Regression classifier
clf = AdaBoostClassifier(n_estimators=1000, random_state=0)
clf.fit(X_train, y_train)

# Predict and evaluate
y_pred = clf.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy with Orthogonalized Features: {accuracy}')
```

Output :

```
Accuracy with Orthogonalized Features: 0.9298245614035088
```

Step 5: Generate Relevant Plots

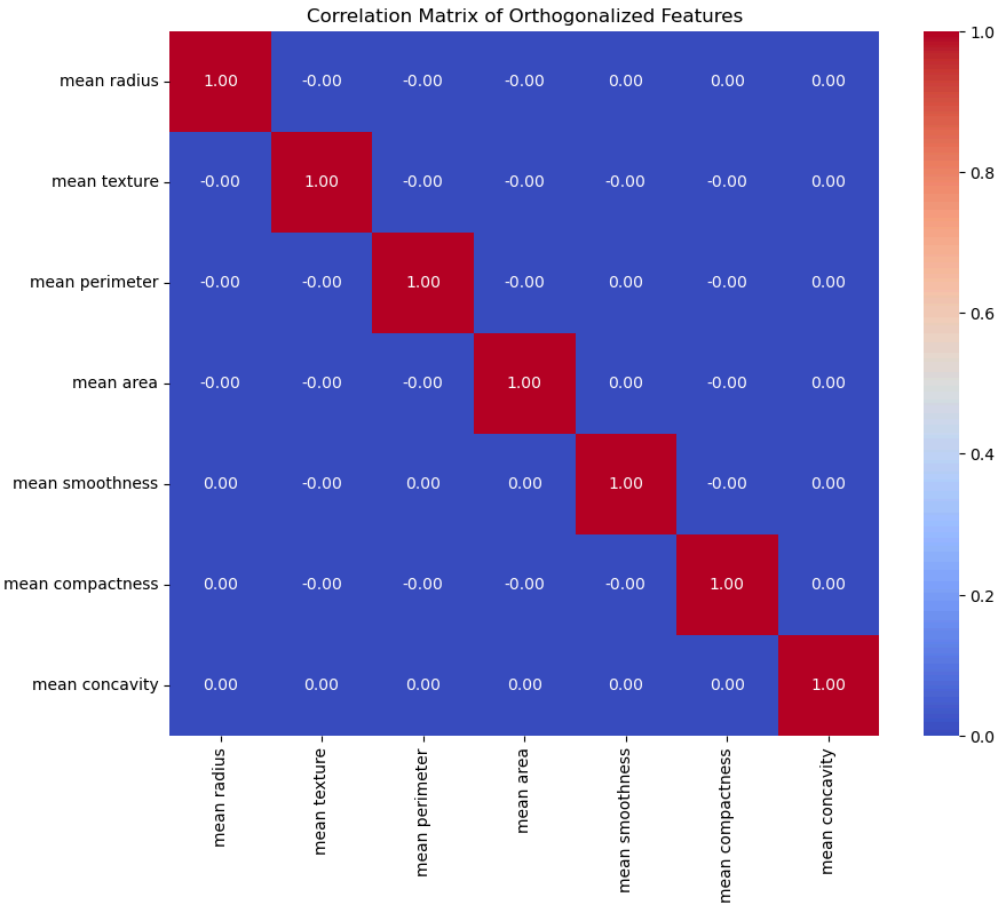
We will generate three plots:

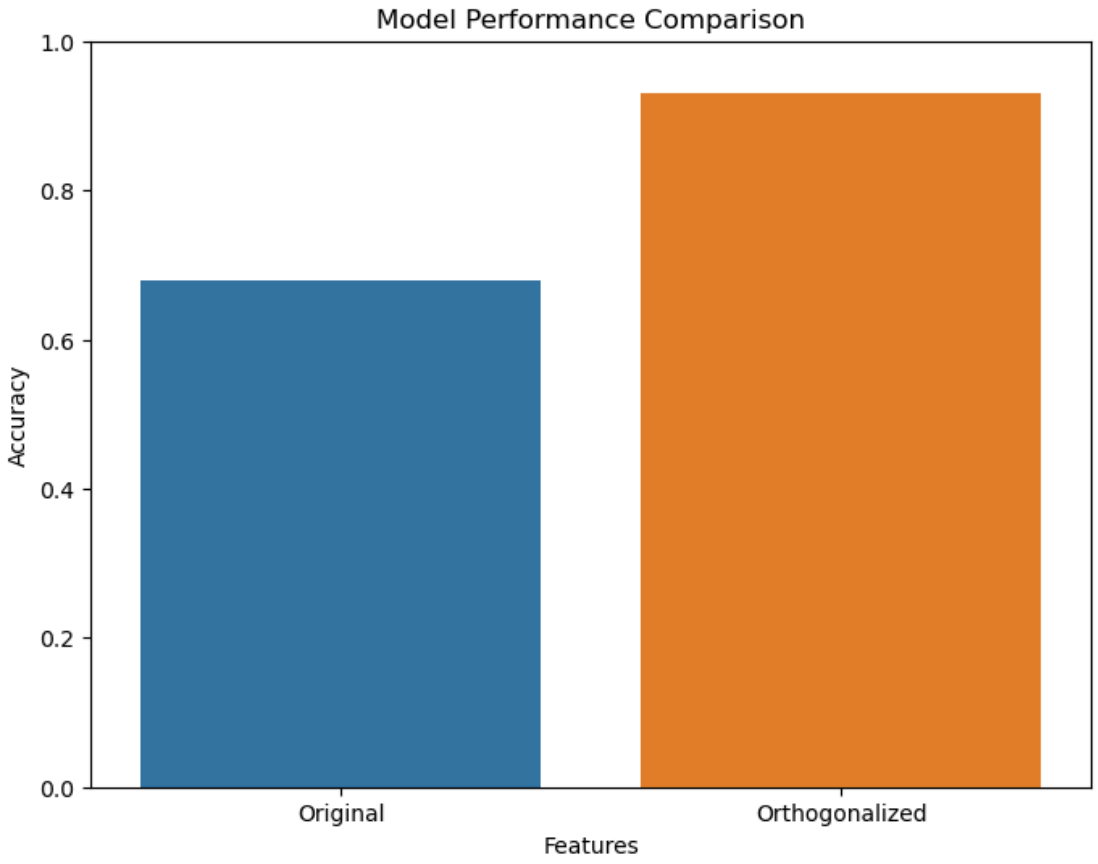
- Correlation Matrix of Original Features
- Correlation Matrix of Orthogonalized Features
- Comparison of Model Performance with Original vs Orthogonalized Features.

```
# Plot 1: Correlation Matrix of Original Features
plt.figure(figsize=(10, 8))
sns.heatmap(df[selected_features].corr(), annot=True, cmap='coolwarm',
fmt='.2f')
plt.title('Correlation Matrix of Original Features')
plt.show()

# Plot 2: Correlation Matrix of Orthogonalized Features
plt.figure(figsize=(10, 8))
sns.heatmap(df_orthogonal[selected_features].corr(), annot=True,
cmap='coolwarm', fmt='.2f')
plt.title('Correlation Matrix of Orthogonalized Features')
plt.show()

# Plot 3: Model Performance Comparison
# Evaluate the model on original features
X_train_orig, X_test_orig, y_train_orig, y_test_orig =
train_test_split(X_normalized, y, test_size=0.3, random_state=42)
clf_orig = LogisticRegression(max_iter=10000)
clf_orig.fit(X_train_orig, y_train_orig)
```

```
y_pred_orig = clf_orig.predict(X_test_orig)
accuracy_orig = accuracy_score(y_test_orig, y_pred_orig)

# Bar plot comparison
accuracy_data = pd.DataFrame({
    'Features': ['Original', 'Orthogonalized'],
    'Accuracy': [accuracy_orig, accuracy]
})

plt.figure(figsize=(8, 6))
sns.barplot(x='Features', y='Accuracy', data=accuracy_data)
plt.title('Model Performance Comparison')
plt.ylim(0, 1)
plt.show()
```

Outputs Generated :





Inference :

Therefore We can notice that, after the orthogonalisation of the considered dataset and performing Classification on the selected " Top K " Attributes of the resultant dataset, there has been a drastic improvement in the Accuracy Performance of the Classification Model from a mere 0.68 to a much appreciated 0.93 %. This Strengthens our Knowledge on the Effects of Orthogonal Transformation on the data.

SVD and Least Square Approximation

Introduction Specificities :

Image compression aims to reduce the amount of data required to represent an image while preserving its visual quality. SVD offers a powerful technique for achieving this. It decomposes the image into its essential components, allowing us to discard less important parts for compression and reconstruct an approximation later.

Least squares approximations - help us quantify the error introduced by discarding information during compression.

The Basic Concept Overview :

Imagine a grayscale image as a matrix `A`. SVD decomposes it into three matrices:

1. **U (Left Singular Vectors):** An orthogonal matrix representing the directions of important variations in the image.
2. **Σ (Singular Values):** A diagonal matrix containing values representing the significance of each direction (U vector) in capturing the image's information. Larger values indicate more importance.

3. **V^T (Right Singular Vectors):** The transpose of another orthogonal matrix, related to how the image data is distributed across the directions in U.

By keeping only the top k largest singular values in Σ and setting the rest to zero, we create a compressed representation A_k of the original image. This captures the most important information with a smaller data size.

A Mathematical Overview :

1. **Singular Value Decomposition (SVD):**

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

where:

- A: Original image matrix (e.g., 4x4 for a 4x4 grayscale image)
- U: Left singular vectors matrix (size same as A)
- Σ: Diagonal matrix containing singular values (size same as A, but only non-zero values on the diagonal)
- V^T: Transpose of right singular vectors matrix (size same as A)

2. Compressed Representation:

$$\Sigma_k = \text{diag}(\sigma_1, \sigma_2, ..., \sigma_k, 0, ..., 0) \quad \text{// Top k singular values}$$
$$A_k = U \Sigma_k V^T$$

where:

- Σ_k: Diagonal matrix containing only the top k singular values from Σ.

- A_k: Compressed representation of the original image.

3. Least Squares Approximation and Error:

Reconstruction: Approximate the original image from the compressed form:

$$A_{k\_approx} = U \Sigma_k V^T$$

- where:
  - A_k_approx: Reconstructed approximation of the original image.

Mean Squared Error (MSE): Quantify the difference between original and reconstructed image:

$$MSE = (1 / (m * n)) * || A - A_{k\_approx} ||^2$$ // m, n are image dimensions

- where:
  - MSE: Mean Squared Error || ||^2: Squared Frobenius norm (measures matrix difference)

By analyzing the MSE for different compression levels (different k values), we can determine the optimal compression ratio that balances image quality and file size.

Pythonic Code :

Step 1: Load and Preprocess the Image

We start by loading the image from a URL and converting it to grayscale using the load_image function. Grayscale conversion is necessary because SVD works on single-channel images.

```
def load_image(url):
    image = io.imread(url)
    if image.ndim == 3:
        image = color.rgb2gray(image)
    return img_as_float(image)
```

Step 2: Compute Singular Value Decomposition (SVD)

```
def compute_svd(image):
    U, S, Vt = np.linalg.svd(image, full_matrices=False)
    return U, S, Vt
```

Step 3: Reconstruct Image Using Top k Singular Values

```
def reconstruct_image(U, S, Vt, k):
    return np.dot(U[:, :k], np.dot(np.diag(S[:k]), Vt[:k, :]))
```

Step 4: Compute the Least Squares Error

```
def compute_error(original, reconstructed):
    return np.linalg.norm(original - reconstructed)
```

Step 5: Compute Explained Variance Ratio

```
def explained_variance_ratio(S, k):
    return np.sum(S[:k]**2) / np.sum(S**2)
```

Step 6: Display and Save Results

```
def display_and_save_results(original, reconstructed, k, error,
explained_variance):
    fig, axes = plt.subplots(1, 2, figsize=(10, 5))

    # Display original image
    axes[0].imshow(original, cmap='gray')
    axes[0].set_title('Original Image')
    axes[0].axis('off')

    # Display reconstructed image
    axes[1].imshow(reconstructed, cmap='gray')
    axes[1].set_title(f'Reconstructed Image (k={k})')
    axes[1].axis('off')

    fig.suptitle(f'k={k}, Error={error:.2f}, Explained
Variance={explained_variance:.2%}')
    plt.show()

    # Save the figure
    fig.savefig(f'result_k_{k}.jpg')
```

```
    plt.close(fig)

    # Save the reconstructed image as a jpg file
    imageio.imwrite(f'reconstructed_image_k_{k}.jpg', (reconstructed *
255).astype(np.uint8))
```

Step 7: Plot Metrics

```
def plot_metrics(singular_values, errors, explained_variances):
    fig, ax1 = plt.subplots()

    color = 'tab:blue'
    ax1.set_xlabel('k (number of singular values)')
    ax1.set_ylabel('Error', color=color)
    ax1.plot(singular_values, errors, color=color, marker='o', linestyle='-',
label='Error')
    ax1.tick_params(axis='y', labelcolor=color)

    ax2 = ax1.twinx()  # instantiate a second axes that shares the same x-axis

    color = 'tab:orange'
    ax2.set_ylabel('Explained Variance Ratio', color=color)  # we already handled
the x-label with ax1
    ax2.plot(singular_values, explained_variances, color=color, marker='o',
linestyle='-', label='Explained Variance Ratio')
    ax2.tick_params(axis='y', labelcolor=color)

    fig.tight_layout()  # otherwise the right y-label is slightly clipped
    plt.title('Error and Explained Variance Ratio vs k')
    plt.show()
```

Outputs Generated :

k=5, Error=61.44, Explained Variance=99.49%

Original Image

Reconstructed Image (k=5)

k=20, Error=32.64, Explained Variance=99.86%

Original Image

Reconstructed Image (k=20)

k=50, Error=15.83, Explained Variance=99.97%

Original Image

Reconstructed Image (k=50)

k=100, Error=6.27, Explained Variance=99.99%

Original Image

Reconstructed Image (k=100)

k=200, Error=0.74, Explained Variance=100.00%

Original Image

Reconstructed Image (k=200)

k=300, Error=0.01, Explained Variance=100.00%

Original Image

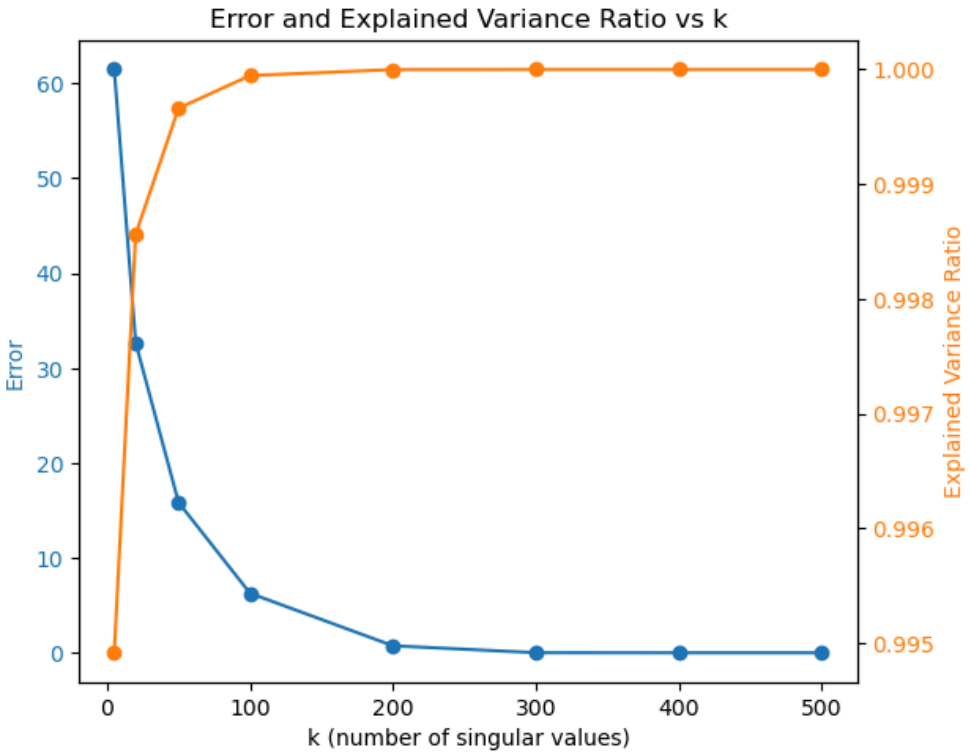
Reconstructed Image (k=300)

k=400, Error=0.00, Explained Variance=100.00%

Original Image

Reconstructed Image (k=400)

Inference :



Inference from the Dual Line Plot

The dual line plot shows the relationship between the number of singular values (k), the reconstruction error, and the explained variance ratio. Here's a detailed analysis:

Error Analysis:

The reconstruction error (blue line) starts high when k is low and decreases rapidly as k increases.
The error drops significantly between k = 5 and k = 100.
After k = 100, the error decreases more gradually and stabilizes around a very low value.

Explained Variance Ratio:

The explained variance ratio (orange line) increases sharply at the beginning.
By k = 50, the explained variance ratio is already very close to 1 (almost 99.9%).
After k = 100, the explained variance ratio changes very little, indicating that most of the variance in the image is already captured.

Conclusion on When to Stop

To achieve a good balance between image quality and compression, we should choose a k where the error is low and the explained variance ratio is high.

From the plot, k = 50 captures almost all the variance (99.9%) while significantly reducing the error. This suggests that k = 50 provides a good compressed image with minimal visual loss.
Visual Loss and Compression:

Beyond k = 100, the improvement in image quality becomes marginal as the error reduction is minimal, and the explained variance ratio is almost constant.
Therefore, using more than 100 singular values does not substantially improve the visual quality but increases the computational cost and storage size.