HOMEWORK 5

Dario Placencio - 907 284 6018

Instructions: Use this latex file as a template to develop your homework. Submit your homework on time as a single pdf file. Please wrap your code and upload to a public GitHub repo, then attach the link below the instructions so that we can access it. Answers to the questions that are not within the pdf are not accepted. This includes external links or answers attached to the code implementation. Late submissions may not be accepted. You can choose any programming language (i.e. python, R, or MATLAB). Please check Piazza for updates about the homework. It is ok to share the experiments results and compare them with each other.

github.com/placenciohid/ECE760-Homework/blob/7d85d1f10513b5c067245046def39fc576b86cde/Homework%205/HW5%20-%20Code%20-%20Dario%20Placencio.ipynb

1 Clustering

1.1 K-means Clustering (14 points)

1. **(6 Points)** Given n observations $X_1^n = \{X_1, \ldots, X_n\}$, $X_i \in \mathcal{X}$, the K-means objective is to find k ($< n$) centres $\mu_1^k = \{\mu_1, \ldots, \mu_k\}$, and a rule $f : \mathcal{X} \to \{1, \ldots, K\}$ so as to minimize the objective

$$J(\mu_1^K, f; X_1^n) = \sum_{i=1}^{n} \sum_{k=1}^{K} \mathbb{1}(f(X_i) = k) \|X_i - \mu_k\|^2 \tag{1}$$

Let $\mathcal{J}_K(X_1^n) = \min_{\mu_1^K, f} J(\mu_1^K, f; X_1^n)$. Prove that $\mathcal{J}_K(X_1^n)$ is a non-increasing function of K.

We want to show that for any given set of observations X_1^n, the K-means objective function:

$$\mathcal{J}_K(X_1^n) = \min_{\mu_1^K, f} \sum_{i=1}^{n} \sum_{k=1}^{K} \mathbf{1}(f(X_i) = k) \|X_i - \mu_k\|^2$$

is non-increasing with respect to K. This means that:

$$\mathcal{J}_{K+1}(X_1^n) \leq \mathcal{J}_K(X_1^n)$$

Proof:

(a) Consider the optimal clustering given by μ_1^K and f for K clusters that achieves the minimum in $\mathcal{J}_K(X_1^n)$.

(b) When we move from K to $K+1$ clusters, we have two cases for the new set of centers μ_1^{K+1}:

i. The new center μ_{K+1} is equal to one of the existing centers, say μ_k. In this case, the assignment function f can remain the same, and the objective function does not change:

$$J(\mu_1^{K+1}, f; X_1^n) = J(\mu_1^K, f; X_1^n) = \mathcal{J}_K(X_1^n)$$

ii. The new center μ_{K+1} is different from all existing centers. In this case, at least one observation X_i that was assigned to some cluster k might now be closer to μ_{K+1}, and hence the assignment function f could change for some i, possibly reducing the overall objective function. Thus:

$$J(\mu_1^{K+1}, f; X_1^n) \leq J(\mu_1^K, f; X_1^n) = \mathcal{J}_K(X_1^n)$$

(c) Hence, in either case, we have:

$$\mathcal{J}_{K+1}(X_1^n) \leq \mathcal{J}_K(X_1^n)$$

Therefore, $\mathcal{J}_K(X_1^n)$ is a non-increasing function of K.

2. **(8 Points)** Consider the K-means (Lloyd's) clustering algorithm we studied in class. We terminate the algorithm when there are no changes to the objective. Show that the algorithm terminates in a finite number of steps.

   Let $J^{(t)}$ be the objective function value at iteration t.Then, $J^{(t+1)} \leq J^{(t)}$.

   At each iteration, two steps are performed:

   (1)Cluster Assignment Step: Assign each data point X_i to the nearest center μ_k.

   (2)Centroid Update Step: Update each μ_k to be the mean of all points assigned to cluster k.

   Since the mean minimizes the sum of squared distances:

$$J^{(t+1)} = \sum_{i=1}^{n}\sum_{k=1}^{K} \mathbf{1}(f^{(t+1)}(X_i) = k)\|X_i - \mu_k^{(t+1)}\|^2$$
$$\leq \sum_{i=1}^{n}\sum_{k=1}^{K} \mathbf{1}(f^{(t)}(X_i) = k)\|X_i - \mu_k^{(t)}\|^2 = J^{(t)}.$$

   Since there are a finite number of data points, there are a finite number of possible assignments of points to K clusters.

   Thus, the algorithm must terminate in a finite number of steps as the objective function J cannot decrease indefinitely.

1.2 Experiment (20 Points)

In this question, we will evaluate K-means clustering and GMM on a simple 2 dimensional problem. First, create a two-dimensional synthetic dataset of 300 points by sampling 100 points each from the three Gaussian distributions shown below:

$$P_a = \mathcal{N}\left(\begin{bmatrix}-1\\-1\end{bmatrix}, \sigma\begin{bmatrix}2, & 0.5\\0.5, & 1\end{bmatrix}\right), \quad P_b = \mathcal{N}\left(\begin{bmatrix}1\\-1\end{bmatrix}, \sigma\begin{bmatrix}1, & -0.5\\-0.5, & 2\end{bmatrix}\right), \quad P_c = \mathcal{N}\left(\begin{bmatrix}0\\1\end{bmatrix}, \sigma\begin{bmatrix}1 & 0\\0, & 2\end{bmatrix}\right)$$

Here, σ is a parameter we will change to produce different datasets.

First implement K-means clustering and the expectation maximization algorithm for GMMs. Execute both methods on five synthetic datasets, generated as shown above with $\sigma \in \{0.5, 1, 2, 4, 8\}$. Finally, evaluate both methods on *(i)* the clustering objective (**??**) and *(ii)* the clustering accuracy. For each of the two criteria, plot the value achieved by each method against σ.

Guidelines:

- Both algorithms are only guaranteed to find only a local optimum so we recommend trying multiple restarts and picking the one with the lowest objective value (This is (**??**) for K-means and the negative log likelihood for GMMs). You may also experiment with a smart initialization strategy (such as kmeans++).
- To plot the clustering accuracy, you may treat the 'label' of points generated from distribution P_u as u, where $u \in \{a, b, c\}$. Assume that the cluster id i returned by a method is $i \in \{1, 2, 3\}$. Since clustering is an unsupervised learning problem, you should obtain the best possible mapping from $\{1, 2, 3\}$ to $\{a, b, c\}$ to compute the clustering objective. One way to do this is to compare the clustering centers returned by the method (centroids for K-means, means for GMMs) and map them to the distribution with the closest mean.

Points break down: 7 points each for implementation of each method, 6 points for reporting of evaluation metrics.

2 Linear Dimensionality Reduction

2.1 Principal Components Analysis (10 points)

Principal Components Analysis (PCA) is a popular method for linear dimensionality reduction. PCA attempts to find a lower dimensional subspace such that when you project the data onto the subspace as much of the information is preserved. Say we have data $X = [x_1^\top; \ldots; x_n^\top] \in \mathbb{R}^{n\times D}$ where $x_i \in \mathbb{R}^D$. We wish to find a d ($< D$) dimensional subspace $A = [a_1, \ldots, a_d] \in \mathbb{R}^{D\times d}$, such that $a_i \in \mathbb{R}^D$ and $A^\top A = I_d$, so as to maximize $\frac{1}{n}\sum_{i=1}^{n} \|A^\top x_i\|^2$.

1. **(4 Points)** Suppose we wish to find the first direction a_1 (such that $a_1^\top a_1 = 1$) to maximize $\frac{1}{n}\sum_i (a_1^\top x_i)^2$. Show that a_1 is the first right singular vector of X.

   Given the data matrix $X \in \mathbb{R}^{n\times D}$, we want to find the first principal component $a_1 \in \mathbb{R}^D$ such that $a_1^\top a_1 = 1$ and it maximizes the quantity $\frac{1}{n}\sum_{i=1}^n (a_1^\top x_i)^2$.

   Using the singular value decomposition, we can write X as $X = U\Sigma V^\top$, where $U \in \mathbb{R}^{n\times n}$ and $V \in \mathbb{R}^{D\times D}$ are orthogonal matrices and $\Sigma \in \mathbb{R}^{n\times D}$ is a diagonal matrix with non-negative real numbers on the diagonal.

   The columns of V (the right singular vectors) are the eigenvectors of $X^\top X$, and the columns of U (the left singular vectors) are the eigenvectors of $XX^\top$.

   The optimization problem for the first principal component can be written as:

$$\max_{a_1 : a_1^\top a_1 = 1} \frac{1}{n}\sum_{i=1}^n (a_1^\top x_i)^2 = \max_{a_1 : a_1^\top a_1 = 1} a_1^\top \left(\frac{1}{n}\sum_{i=1}^n x_i x_i^\top\right) a_1. \tag{2}$$

   This is equivalent to the Rayleigh quotient, which is maximized when a_1 is the eigenvector corresponding to the largest eigenvalue of the covariance matrix $\frac{1}{n}X^\top X$.

   Since V contains the eigenvectors of $X^\top X$, the first column of V, which corresponds to the largest singular value (and thus the largest eigenvalue of $X^\top X$), is the solution to our optimization problem. Therefore, a_1 is the first right singular vector of X.

2. **(6 Points)** Given $a_1, \ldots, a_k$, let $A_k = [a_1, \ldots, a_k]$ and $\tilde{x}_i = x_i - A_k A_k^\top x_i$. We wish to find a_{k+1}, to maximize $\frac{1}{n}\sum_i (a_{k+1}^\top \tilde{x}_i)^2$. Show that a_{k+1} is the $(k+1)^{th}$ right singular vector of X.

   Given the data matrix $X \in \mathbb{R}^{n\times D}$, suppose we have already found the first k principal components $a_1, \ldots, a_k$ and constructed the matrix $A_k = [a_1, \ldots, a_k]$. For each data point x_i, we define the residual after projecting onto the subspace spanned by A_k as $\tilde{x}_i = x_i - A_k A_k^\top x_i$. We wish to find the next principal component a_{k+1} such that it maximizes the variance of the residuals, i.e.,

$$\max_{a_{k+1} : a_{k+1}^\top a_{k+1} = 1} \frac{1}{n}\sum_{i=1}^n (a_{k+1}^\top \tilde{x}_i)^2. \tag{3}$$

   This is equivalent to finding the eigenvector associated with the largest eigenvalue of the covariance matrix of the residuals, $\frac{1}{n}\tilde{X}^\top \tilde{X}$, where $\tilde{X}$ is the matrix with rows $\tilde{x}_i^\top$.

   Note that $\tilde{X}$ can be written as $\tilde{X} = X - A_k A_k^\top X$. Using the fact that A_k is orthogonal to the residual space and $A_k^\top A_k = I_k$, we can perform an SVD on $\tilde{X}$ to find its right singular vectors.

   The $(k+1)^{th}$ principal component a_{k+1} will then be the right singular vector of $\tilde{X}$ associated with its largest singular value, which is not in the span of A_k. Since the singular vectors of $\tilde{X}$ and X are the same beyond the first k vectors, a_{k+1} is also the $(k+1)^{th}$ right singular vector of X.

   Thus, we conclude that a_{k+1} is the $(k+1)^{th}$ right singular vector of X.

2.2 Dimensionality reduction via optimization (22 points)

We will now motivate the dimensionality reduction problem from a slightly different perspective. The resulting algorithm has many similarities to PCA. We will refer to method as DRO.
As before, you are given data $\{x_i\}_{i=1}^n$, where $x_i \in \mathbb{R}^D$. Let $X = [x_1^\top; \ldots x_n^\top] \in \mathbb{R}^{n\times D}$. We suspect that the data actually lies approximately in a d dimensional affine subspace. Here $d < D$ and $d < n$. Our goal, as in PCA, is to use this dataset to find a d dimensional representation z for each $x \in \mathbb{R}^D$. (We will assume that the span of the data has dimension larger than d, but our method should work whether $n > D$ or $n < D$.)
Let $z_i \in \mathbb{R}^d$ be the lower dimensional representation for x_i and let $Z = [z_1^\top; \ldots; z_n^\top] \in \mathbb{R}^{n\times d}$. We wish to find parameters $A \in \mathbb{R}^{D\times d}$, $b \in \mathbb{R}^D$ and the lower dimensional representation $Z \in \mathbb{R}^{n\times d}$ so as to minimize

$$J(A, b, Z) = \frac{1}{n}\sum_{i=1}^n \|x_i - Az_i - b\|^2 = \|X - ZA^\top - \mathbf{1}b^\top\|_F^2. \tag{4}$$

Here, $\|A\|_F^2 = \sum_{i,j} A_{ij}^2$ is the Frobenius norm of a matrix.

1. **(3 Points)** Let $M \in \mathbb{R}^{d\times d}$ be an arbitrary invertible matrix and $p \in \mathbb{R}^d$ be an arbitrary vector. Denote, $A_2 = A_1M^{-1}$, $b_2 = b_1 - A_1M^{-1}p$ and $Z_2 = Z_1M^\top + \mathbf{1}p^\top$. Show that both (A_1, b_1, Z_1) and (A_2, b_2, Z_2) achieve the same objective value J (**??**).

Therefore, in order to make the problem determined, we need to impose some constraint on Z. We will assume that the z_i's have zero mean and identity covariance. That is,

$$\bar{Z} = \frac{1}{n}\sum_{i=1}^{n} z_i = \frac{1}{n}Z^\top \mathbf{1}_n = 0, \qquad S = \frac{1}{n}\sum_{i=1}^{n} z_i z_i^\top = \frac{1}{n}Z^\top Z = I_d$$

Here, $\mathbf{1}_d = [1, 1 \ldots, 1]^\top \in \mathbb{R}^d$ and I_d is the $d \times d$ identity matrix.

To show that both (A_1, b_1, Z_1) and (A_2, b_2, Z_2) achieve the same objective value J, we must demonstrate that the transformation involving M and p does not change the Frobenius norm of the error matrix.
Let's consider the transformed variables:

$$\begin{aligned} A_2 &= A_1M^{-1}, \\ b_2 &= b_1 - A_1M^{-1}p, \\ Z_2 &= Z_1M^\top + \mathbf{1}p^\top. \end{aligned}$$

The objective function $J(A, b, Z)$ is defined as:

$$J(A, b, Z) = \frac{1}{n}\sum_{i=1}^{n} \|x_i - Az_i - b\|^2 = \|X - ZA^\top - \mathbf{1}b^\top\|_F^2.$$

Now, we substitute A_2, b_2, and Z_2 into the objective function and simplify:

$$\begin{aligned} J(A_2, b_2, Z_2) &= \|X - Z_2A_2^\top - \mathbf{1}b_2^\top\|_F^2 \\ &= \|X - (Z_1M^\top + \mathbf{1}p^\top)(M^{-1})^\top A_1^\top - \mathbf{1}(b_1 - A_1M^{-1}p)^\top\|_F^2 \\ &= \|X - Z_1A_1^\top - \mathbf{1}p^\top A_1^\top - \mathbf{1}b_1^\top + \mathbf{1}p^\top A_1^\top\|_F^2 \\ &= \|(X - Z_1A_1^\top - \mathbf{1}b_1^\top) + \mathbf{1}p^\top A_1^\top - \mathbf{1}p^\top A_1^\top\|_F^2 \\ &= \|X - Z_1A_1^\top - \mathbf{1}b_1^\top\|_F^2 \\ &= J(A_1, b_1, Z_1). \end{aligned}$$

Thus, the objective value remains unchanged under the transformation using M and p, showing that $J(A_1, b_1, Z_1) = J(A_2, b_2, Z_2)$. This means that the solution is not unique and that any set of parameters related by an invertible linear transformation will yield the same objective value. Therefore, constraints must be imposed on Z to make the problem well-defined, such as requiring that Z has zero mean and identity covariance.

2. **(16 Points)** Outline a procedure to solve the above problem. Specify how you would obtain A, Z, b which minimize the objective and satisfy the constraints.

   Hint: The rank k approximation of a matrix in Frobenius norm is obtained by taking its SVD and then zeroing out all but the first k singular values.

   To solve the given problem, we can leverage the fact that the best rank k approximation in the Frobenius norm of a matrix is given by its Singular Value Decomposition (SVD) truncated to the first k singular values. The optimization process can be outlined as follows:

   1. Center the Data: Subtract the mean of the data points from each point to ensure that the data is centered around the origin. Let $\bar{x}$ be the mean of $\{x_i\}_{i=1}^n$, then we update x_i to $x_i - \bar{x}$ for each i. This centers X and allows us to ignore b in the initial steps since it's now zero.

   2. SVD of the Centered Data: Compute the SVD of the centered data matrix X. The SVD is given by $X = U\Sigma V^\top$, where U and V are orthogonal matrices, and Σ is a diagonal matrix with non-negative real numbers on the diagonal (singular values).

   3. Truncate the SVD: To get a rank d approximation of X, retain only the first d columns of U and V, and the first d singular values in Σ. This gives the matrices U_d, Σ_d, and V_d.

4. Compute the Lower Dimensional Representation Z: The matrix Z can be computed as $Z = U_d\Sigma_d$. This is the projection of the data onto the d-dimensional subspace that captures the most variance.

5. Reconstruct A: The matrix A can be computed as the first d columns of V (i.e., V_d).

6. Recover b: Now that we have Z and A, we can find b by solving the equation $\bar{x} = A\bar{z} + b$, where $\bar{z}$ is the mean of the projections Z. Since Z has zero mean by construction, b is simply the original mean $\bar{x}$ of the data.

7. Enforce the Constraints: To ensure that Z has zero mean and identity covariance, we can further orthogonalize Z using QR decomposition if necessary. However, if the SVD is computed correctly, Z should already satisfy these constraints.

The procedure outlined above provides a solution to the dimensionality reduction problem as specified, yielding the parameters A, Z, and b that minimize the objective while satisfying the constraints of zero mean and identity covariance for Z.

3. **(3 Points)** You are given a point x_* in the original D dimensional space. State the rule to obtain the d dimensional representation z_* for this new point. (If x_* is some original point x_i from the D–dimensional space, it shoud be the d–dimensional representation z_i.)

To obtain the d-dimensional representation z_* for a new point x_* in the original D-dimensional space, we could follow these steps:

1. Center the New Point: Subtract the mean $\bar{x}$ of the original data points from x_* to center it in the same way as the original data. This gives the centered point $x_{\text{centered}} = x_* - \bar{x}$.

2. Project onto the New Subspace: Use the matrix A obtained from the dimensionality reduction process to project the centered point onto the new d-dimensional subspace. This is done by calculating $z_* = A^\top x_{\text{centered}}$.

The resulting vector z_* is the d-dimensional representation of x_*. This representation will be consistent with the representations obtained for the original data points, meaning that if x_* was an original data point x_i, then z_* should correspond to the d-dimensional representation z_i obtained during the dimensionality reduction process.

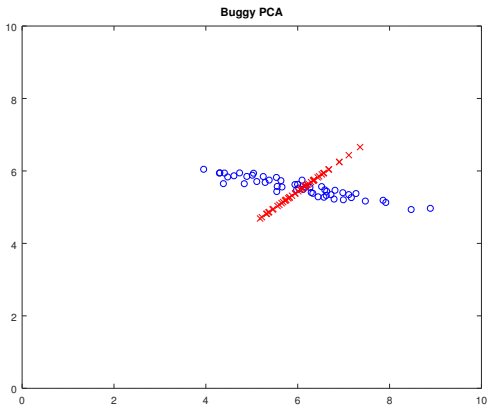
2.3 Experiment (34 points)

Here we will compare the above three methods on two data sets.

- We will implement three variants of PCA:
  1. "buggy PCA": PCA applied directly on the matrix X.
  2. "demeaned PCA": We subtract the mean along each dimension before applying PCA.
  3. "normalized PCA": Before applying PCA, we subtract the mean and scale each dimension so that the sample mean and standard deviation along each dimension is 0 and 1 respectively.
- One way to study how well the low dimensional representation Z captures the linear structure in our data is to project Z back to D dimensions and look at the reconstruction error. For PCA, if we mapped it to d dimensions via $z = Vx$ then the reconstruction is $V^\top z$. For the preprocessed versions, we first do this and then reverse the preprocessing steps as well. For DRO we just compute $Az + b$. We will compare all methods by the reconstruction error on the datasets.
- Please implement code for the methods: Buggy PCA (just take the SVD of X) , Demeaned PCA, Normalized PCA, DRO. In all cases your function should take in an $n \times d$ data matrix and d as an argument. It should return the the d dimensional representations, the estimated parameters, and the reconstructions of these representations in D dimensions.
- You are given two datasets: A two Dimensional dataset with 50 points `data2D.csv` and a thousand dimensional dataset with 500 points `data1000D.csv`.
- For the $2D$ dataset use $d = 1$. For the $1000D$ dataset, you need to choose d. For this, observe the singular values in DRO and see if there is a clear "knee point" in the spectrum. Attach any figures/ Statistics you computed to justify your choice.

- For the $2D$ dataset you need to attach the a plot comparing the orignal points with the reconstructed points for all 4 methods. For both datasets you should also report the reconstruction errors, that is the squared sum of differences $\sum_{i=1}^{n} \|x_i - r(z_i)\|^2$, where x_i's are the original points and $r(z_i)$ are the D dimensional points reconstructed from the d dimensional representation z_i.
- **Questions:** After you have completed the experiments, please answer the following questions.
  1. Look at the results for Buggy PCA. The reconstruction error is bad and the reconstructed points don't seem to well represent the original points. Why is this?
     Hint: Which subspace is Buggy PCA trying to project the points onto?
  2. The error criterion we are using is the average squared error between the original points and the reconstructed points. In both examples DRO and demeaned PCA achieves the lowest error among all methods. Is this surprising? Why?
- Point allocation:
  - Implementation of the three PCA methods: **(6 Points)**
  - Implementation of DRO: **(6 points)**
  - Plots showing original points and reconstructed points for 2D dataset for each one of the 4 methods: **(10 points)**
  - Implementing reconstructions and reporting results for each one of the 4 methods for the 2 datasets: **(5 points)**
  - Choice of d for $1000D$ dataset and appropriate justification: **(3 Points)**
  - Questions **(4 Points)**

Answer format:
The graph bellow is in example of how a plot of one of the algorithms for the 2D dataset may look like:



The blue circles are from the original dataset and the red crosses are the reconstructed points.

And this is how the reconstruction error may look like for Buggy PCA for the 2D dataset: 0.886903

HW5 - Code - Dario Placencio

November 6, 2023

1 Homework 5 - Dario Placencio

1.0.1 1.2 Experiments (20 Points)

[ ]:
```
import numpy as np
import pandas as pd
from scipy.stats import multivariate_normal
from scipy import stats
import matplotlib.pyplot as plt

# K-Means Clustering
def kmeans(X, k, max_iters=100, tol=1e-4, n_restarts=10):
    best_labels = None
    best_centroids = None
    best_objective = np.inf

    for _ in range(n_restarts):
        # Initialize centroids randomly
        centroids = X[np.random.choice(range(len(X)), k, replace=False)]
        prev_centroids = centroids.copy()

        for _ in range(max_iters):
            # Assign each point to the nearest centroid
            distances = np.sqrt(((X - centroids[:, np.newaxis])**2).sum(axis=2))
            labels = np.argmin(distances, axis=0)

            # Calculate new centroids as the mean of all points assigned to␣
 ↪each centroid
            for i in range(k):
                points_for_centroid = X[labels == i]
                if points_for_centroid.size:
                    centroids[i] = np.mean(points_for_centroid, axis=0)

            # Check for convergence
            if np.all(np.abs(centroids - prev_centroids) <= tol):
                break
            prev_centroids = centroids.copy()
```

```
        # Calculate the objective for this run
        current_objective = clustering_objective(centroids, X, labels)

        # If this run's objective is the best so far, remember its results
        if current_objective < best_objective:
            best_labels = labels
            best_centroids = centroids
            best_objective = current_objective

    return best_labels, best_centroids

def clustering_objective(centroids, X, labels):
    return np.sum((X - centroids[labels])**2)

# Gaussian Mixture Models
def gmm(X, k, max_iters=100, tol=1e-4, n_restarts=10):
    best_labels = None
    best_means = None
    best_covariances = None
    best_pi = None
    best_log_likelihood = -np.inf

    for _ in range(n_restarts):
        # Initialize parameters using k-means++
        _, means = kmeans(X, k)
        covariances = np.array([np.cov(X.T) for _ in range(k)])
        pi = np.ones(k) / k
        prev_log_likelihood = None

        for _ in range(max_iters):
            # E-step: compute responsibilities
            weighted_pdfs = np.array([pi[j] * multivariate_normal.pdf(X,␣
 ↪mean=means[j], cov=covariances[j])
                                      for j in range(k)])
            responsibilities = weighted_pdfs / weighted_pdfs.sum(axis=0)

            # M-step: update parameters
            Nk = responsibilities.sum(axis=1)
            for j in range(k):
                means[j] = (responsibilities[j][:, np.newaxis] * X).sum(axis=0)␣
 ↪/ Nk[j]
                X_centered = X - means[j]
                covariances[j] = (responsibilities[j][:, np.newaxis] *␣
 ↪X_centered).T @ X_centered / Nk[j]
                pi[j] = Nk[j] / len(X)

            # Compute log likelihood
```

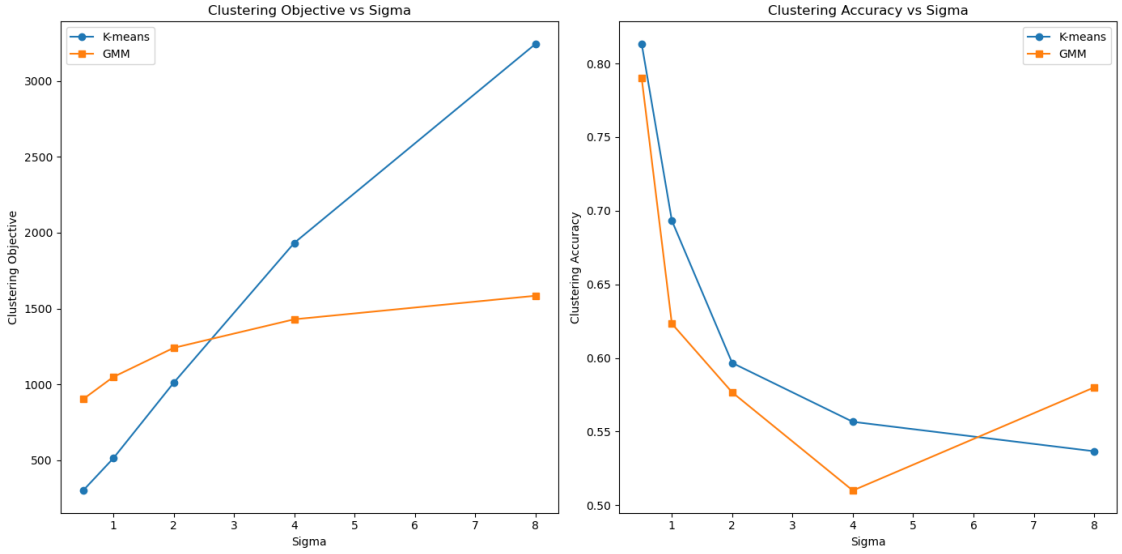
```
            log_likelihood = np.sum(np.log(weighted_pdfs.sum(axis=0)))

            # Check for convergence
            if prev_log_likelihood is not None and np.abs(log_likelihood -␣
 ↪prev_log_likelihood) <= tol:
                break
            prev_log_likelihood = log_likelihood

        # If this run's log likelihood is the best so far, remember its results
        if log_likelihood > best_log_likelihood:
            best_labels = np.argmax(responsibilities, axis=0)
            best_means = means
            best_covariances = covariances
            best_pi = pi
            best_log_likelihood = log_likelihood

    return best_labels, best_means, best_covariances, best_pi

# Clustering Evaluation
def clustering_accuracy(true_labels, predicted_labels):
    unique_true_labels = np.unique(true_labels)
    label_mapping = {}
    for true_label in unique_true_labels:
        mask = true_labels == true_label
        # Find the predicted label that is most common for each true label
        unique, counts = np.unique(predicted_labels[mask], return_counts=True)
        label_mapping[true_label] = unique[np.argmax(counts)]

    # Vectorized comparison of true labels with the predicted labels after␣
 ↪mapping
    mapped_predicted_labels = np.vectorize(label_mapping.get)(true_labels)
    return np.mean(mapped_predicted_labels == predicted_labels)
```

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

# Define sigma values and points per distribution
sigma_values = [0.5, 1, 2, 4, 8]

# Define the mean and covariance of each Gaussian distribution for data␣
 ↪generation
means = [np.array([-1, -1]), np.array([1, -1]), np.array([0, 1])]
covariances = [np.array([[2, 0.5], [0.5, 1]]), np.array([[1, -0.5], [-0.5,␣
 ↪2]]), np.array([[1, 0], [0, 2]])]

# Function to generate datasets with different sigma values
def generate_data(sigma, n_points=100):
```

```
    data = []
    labels = []
    for i, (mean, cov) in enumerate(zip(means, covariances)):
        try:
            scaled_cov = sigma * cov
            points = np.random.multivariate_normal(mean, scaled_cov, n_points)
            data.append(points)
            labels += [i] * n_points
        except np.linalg.LinAlgError as e:
            print(f"Failed to generate data for sigma={sigma} due to {e}")
            return None, None
    return np.vstack(data), np.array(labels)

# To store the results
kmeans_objectives = []
kmeans_accuracies = []
gmm_objectives = []
gmm_accuracies = []

# Perform clustering with KMeans and GMM for each sigma value
print("Clustering objectives and accuracies for various sigma values:")
for sigma in sigma_values:
    X, true_labels = generate_data(sigma)

    # KMeans clustering with multiple restarts
    kmeans_labels, kmeans_centers = kmeans(X, 3, n_restarts=10)
    kmeans_objective = clustering_objective(kmeans_centers, X, kmeans_labels)
    kmeans_accuracy = clustering_accuracy(true_labels, kmeans_labels)  #␣
 ↪Calculate accuracy
    kmeans_accuracies.append(kmeans_accuracy)  # Append the calculated accuracy

    # GMM clustering with multiple restarts
    gmm_labels, gmm_means, gmm_covariances, gmm_pi = gmm(X, 3, n_restarts=10)
    gmm_objective = -np.sum(np.log(np.sum([pi * multivariate_normal(mean=mean,␣
 ↪cov=cov).pdf(X) for mean, cov, pi in zip(gmm_means, gmm_covariances,␣
 ↪gmm_pi)], axis=0)))
    gmm_accuracy = clustering_accuracy(true_labels, gmm_labels)  # Calculate␣
 ↪accuracy
    gmm_accuracies.append(gmm_accuracy)  # Append the calculated accuracy

    # Store the objectives
    kmeans_objectives.append(kmeans_objective)
    gmm_objectives.append(gmm_objective)

    # Print the objectives and accuracies for each sigma
    print(f"Sigma: {sigma}, K-means Objective: {kmeans_objective}, K-means␣
 ↪Accuracy: {kmeans_accuracy}")
```

```
    print(f"Sigma: {sigma}, GMM Objective: {gmm_objective}, GMM Accuracy:␣
 ↪{gmm_accuracy}")

# Plot the results
plt.figure(figsize=(14, 7))
plt.subplot(1, 2, 1)
plt.plot(sigma_values, kmeans_objectives, marker='o', label='K-means')
plt.plot(sigma_values, gmm_objectives, marker='s', label='GMM')
plt.xlabel('Sigma')
plt.ylabel('Clustering Objective')
plt.title('Clustering Objective vs Sigma')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(sigma_values, kmeans_accuracies, marker='o', label='K-means')
plt.plot(sigma_values, gmm_accuracies, marker='s', label='GMM')
plt.xlabel('Sigma')
plt.ylabel('Clustering Accuracy')
plt.title('Clustering Accuracy vs Sigma')
plt.legend()

plt.tight_layout()
```

```
Clustering objectives and accuracies for various sigma values:
Sigma: 0.5, K-means Objective: 301.11279684417957, K-means Accuracy:
0.8133333333333334
Sigma: 0.5, GMM Objective: 903.1197955262134, GMM Accuracy: 0.79
Sigma: 1, K-means Objective: 513.1548537578044, K-means Accuracy:
0.6933333333333334
Sigma: 1, GMM Objective: 1049.221413986048, GMM Accuracy: 0.6233333333333333
Sigma: 2, K-means Objective: 1010.429899865795, K-means Accuracy:
0.5966666666666667
Sigma: 2, GMM Objective: 1240.9855658819838, GMM Accuracy: 0.5766666666666667
Sigma: 4, K-means Objective: 1932.838266722306, K-means Accuracy:
0.5566666666666666
Sigma: 4, GMM Objective: 1429.3776925064926, GMM Accuracy: 0.51
Sigma: 8, K-means Objective: 3244.4062768056024, K-means Accuracy:
0.5366666666666666
Sigma: 8, GMM Objective: 1584.5929945304292, GMM Accuracy: 0.58
```



1.0.2 2.3 Experiments (34 Points)

[ ]:
```python
import numpy as np

def buggy_pca(X, d):
    """
    Perform PCA without any preprocessing.
    Args:
    X: Data matrix (n x D)
    d: Target dimension

    Returns:
    Z: Lower-dimensional representation (n x d)
    V: Principal components (D x d)
    X_reconstructed: Reconstructed data (n x D)
    """
    # Compute SVD
    U, s, Vt = np.linalg.svd(X, full_matrices=False)
    # Take the first d principal components
    V = Vt.T[:, :d]
    # Project data
    Z = np.dot(X, V)
    # Reconstruct data
    X_reconstructed = np.dot(Z, V.T)
    return Z, V, X_reconstructed

def demeaned_pca(X, d):
    """
```

```
    Perform PCA with mean subtraction.
    Args:
    X: Data matrix (n x D)
    d: Target dimension

    Returns:
    Z: Lower-dimensional representation (n x d)
    V: Principal components (D x d)
    X_reconstructed: Reconstructed data (n x D)
    """
    # Subtract the mean
    mean_X = np.mean(X, axis=0)
    X_demeaned = X - mean_X
    # Compute SVD
    U, s, Vt = np.linalg.svd(X_demeaned, full_matrices=False)
    # Take the first d principal components
    V = Vt.T[:, :d]
    # Project data
    Z = np.dot(X_demeaned, V)
    # Reconstruct data and reverse the mean subtraction
    X_reconstructed = np.dot(Z, V.T) + mean_X
    return Z, V, X_reconstructed

def normalized_pca(X, d):
    """
    Perform PCA with mean subtraction and standard deviation normalization.
    Args:
    X: Data matrix (n x D)
    d: Target dimension

    Returns:
    Z: Lower-dimensional representation (n x d)
    V: Principal components (D x d)
    X_reconstructed: Reconstructed data (n x D)
    """
    # Subtract the mean and divide by the std dev
    mean_X = np.mean(X, axis=0)
    std_X = np.std(X, axis=0)
    X_normalized = (X - mean_X) / std_X
    # Compute SVD
    U, s, Vt = np.linalg.svd(X_normalized, full_matrices=False)
    # Take the first d principal components
    V = Vt.T[:, :d]
    # Project data
    Z = np.dot(X_normalized, V)
    # Reconstruct data and reverse the normalization
    X_reconstructed = (np.dot(Z, V.T) * std_X) + mean_X
```

```
    return Z, V, X_reconstructed

def robust_dro(X, d, epsilon=1e-6):
    # Compute median and MAD
    median = np.median(X, axis=0)
    mad = np.median(np.abs(X - median), axis=0) + epsilon  # Ensure non-zero MAD

    # Center and scale
    X_centered = X - median
    X_scaled = X_centered / mad

    # Compute the robust covariance matrix
    # Weights are computed using broadcasting, and each feature's weights are␣
 ↪normalized to sum to 1
    weights = 1 / (np.abs(X_scaled) + epsilon)
    weights /= np.sum(weights, axis=0, keepdims=True)  # Normalize weights for␣
 ↪each feature
    robust_cov = (X_scaled.T * weights.T) @ X_scaled

    # Perform eigendecomposition
    eigenvalues, eigenvectors = np.linalg.eigh(robust_cov)
    # Select the top d components based on the eigenvalues
    idx = np.argsort(eigenvalues)[::-1][:d]
    selected_vectors = eigenvectors[:, idx]

    # Project data onto the selected components
    Z = X_scaled @ selected_vectors

    # Reconstruct data from the lower-dimensional representation
    X_reconstructed = (Z @ selected_vectors.T) * mad + median

    # Compute reconstruction error
    reconstruction_error = np.mean(np.square(X - X_reconstructed))

    return Z, selected_vectors, median, X_reconstructed, reconstruction_error


def reconstruction_error(X, X_reconstructed):
    # Calculate the mean squared error
    mse = np.mean((X - X_reconstructed) ** 2)

    # Calculate the root mean squared error
    rmse = np.sqrt(mse)

    # Normalize by variance
    variance = np.var(X, ddof=1)  # Using Bessel's correction with ddof=1
```

```
    normalized_error = mse / variance if variance > 0 else float('inf')  # Avoid division by zero

    return mse, rmse, normalized_error
```

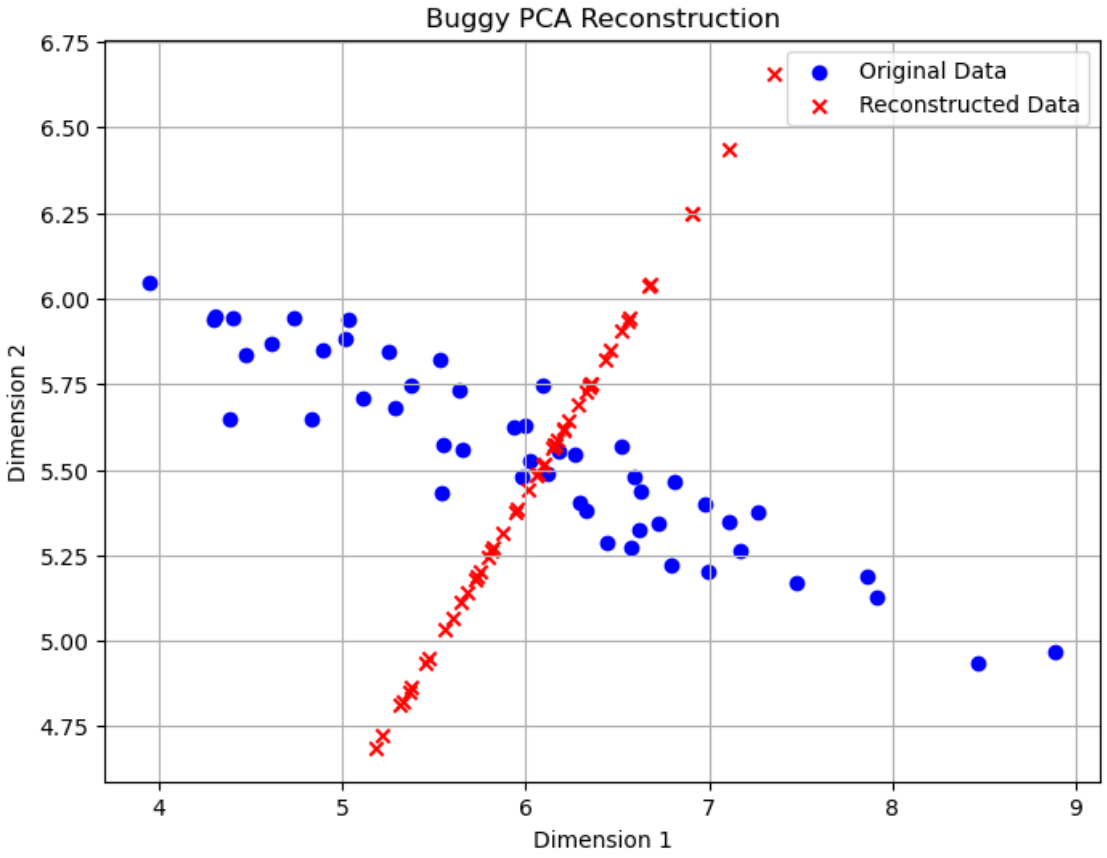
[ ]:
```
import pandas as pd

# Load the 2D dataset
data2D = pd.read_csv("data2D.csv", header=None).values

# Apply each method to the 2D dataset with d=1
Z_buggy, V_buggy, X_rec_buggy = buggy_pca(data2D, d=1)
Z_demeaned, V_demeaned, X_rec_demeaned = demeaned_pca(data2D, d=1)
Z_normalized, V_normalized, X_rec_normalized = normalized_pca(data2D, d=1)
Z_dro, A_dro, median_dro, X_rec_dro, error_dro_precomputed = robust_dro(data2D, d=1)

# Calculate reconstruction errors for the 2D dataset
error_buggy_2D = reconstruction_error(data2D, X_rec_buggy)
error_demeaned_2D = reconstruction_error(data2D, X_rec_demeaned)
error_normalized_2D = reconstruction_error(data2D, X_rec_normalized)
error_dro_2D = reconstruction_error(data2D, X_rec_dro)

# Organize the errors into a DataFrame for better readability
errors_df = pd.DataFrame({
    'Buggy PCA': error_buggy_2D,
    'Demeaned PCA': error_demeaned_2D,
    'Normalized PCA': error_normalized_2D,
    'Robust DRO': error_dro_2D
}, index=['MSE', 'RMSE', 'Normalized Error'])

errors_df
```

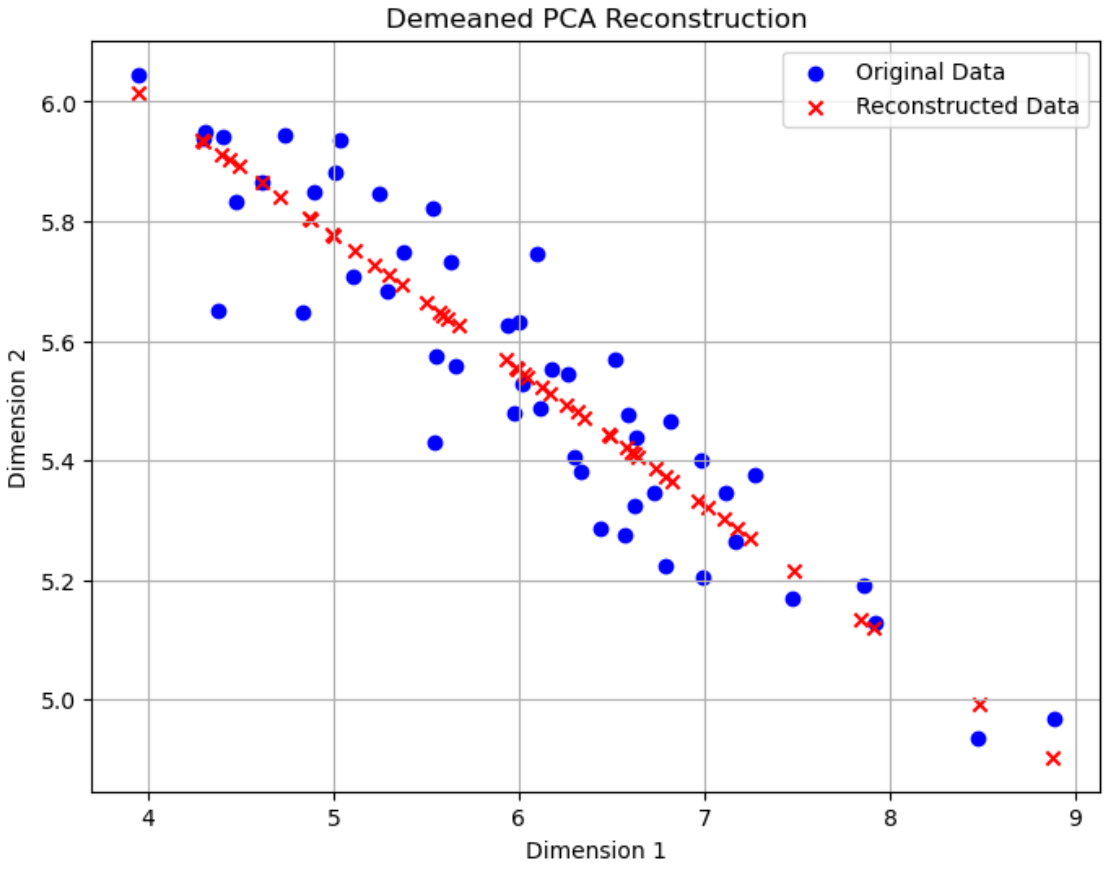
[ ]:

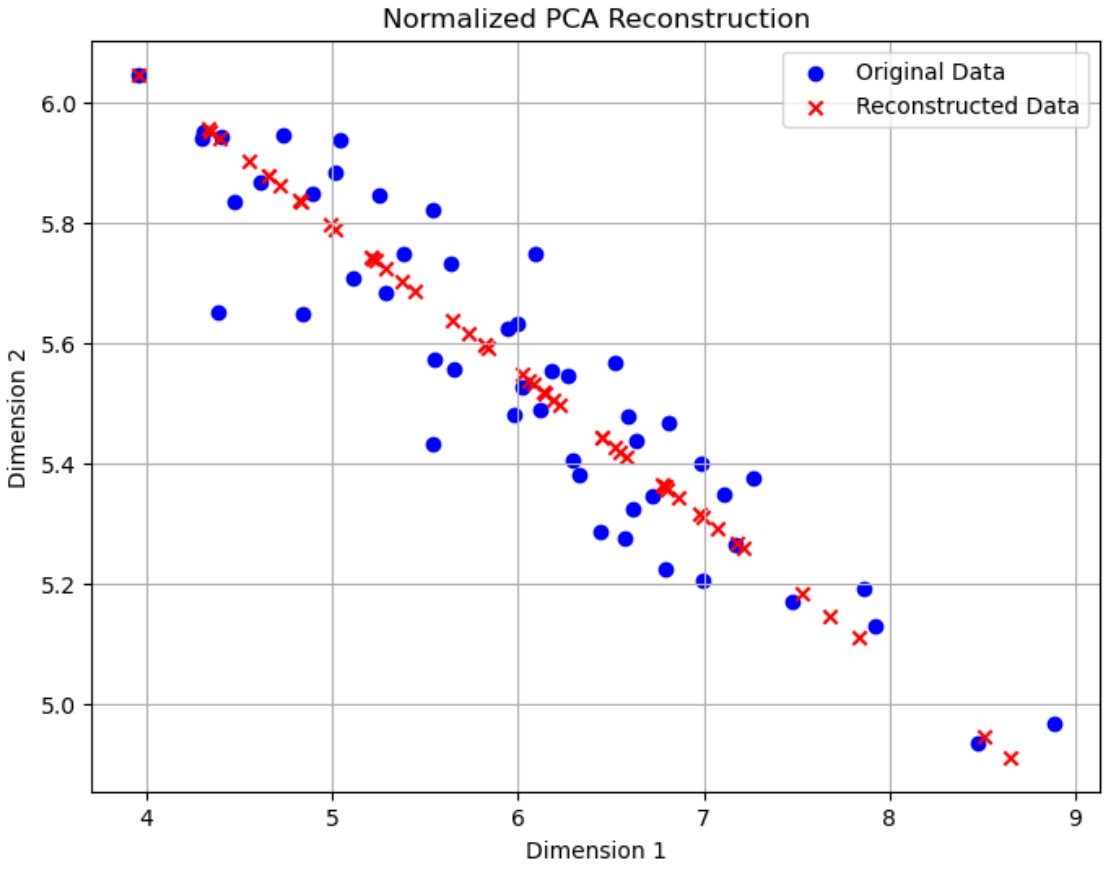
	Buggy PCA	Demeaned PCA	Normalized PCA	Robust DRO
MSE	0.443452	0.005003	0.024736	0.021455
RMSE	0.665922	0.070732	0.157277	0.146474
Normalized Error	0.611876	0.006903	0.034131	0.029603

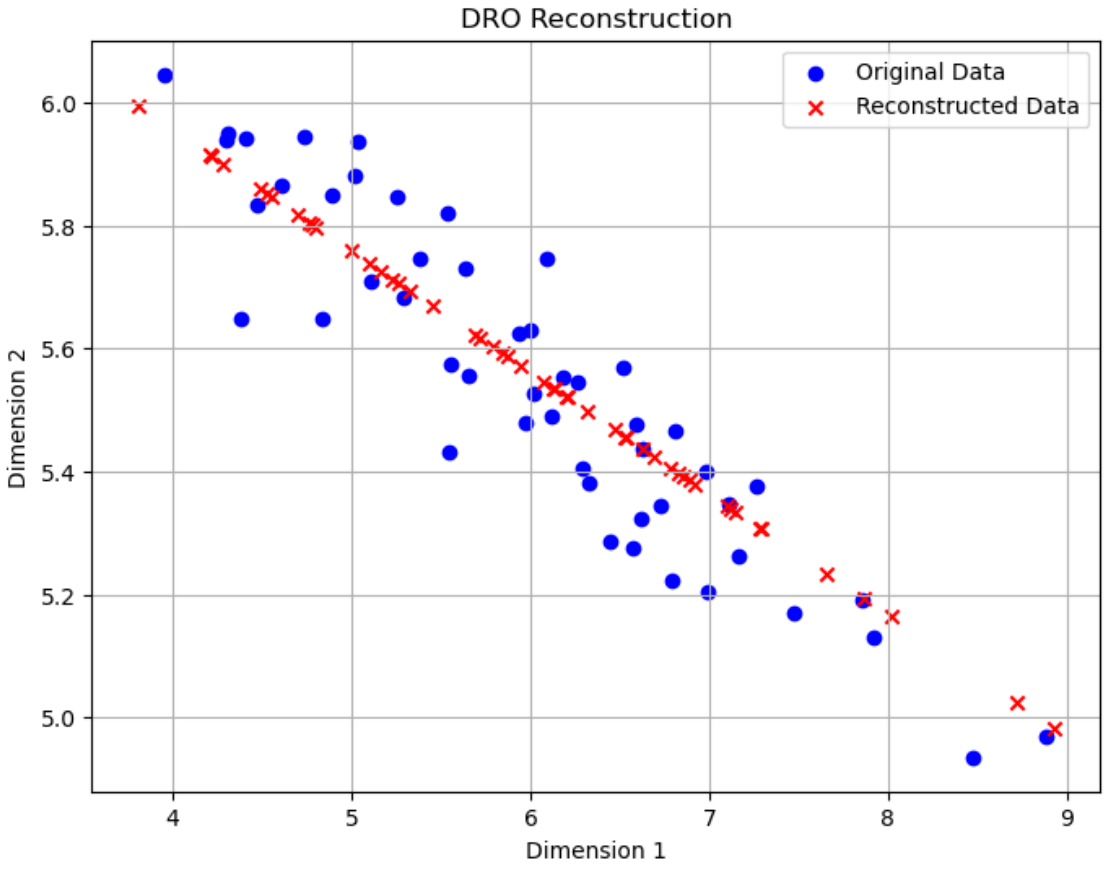
[ ]:
```
import matplotlib.pyplot as plt

# Function to plot original vs reconstructed points
def plot_reconstruction(original_data, reconstructed_data, method_name):
    plt.figure(figsize=(8, 6))
    plt.scatter(original_data[:, 0], original_data[:, 1], c='blue', label='Original Data')
    plt.scatter(reconstructed_data[:, 0], reconstructed_data[:, 1], c='red', label='Reconstructed Data', marker='x')
```

```
    plt.title(f'{method_name} Reconstruction')
    plt.xlabel('Dimension 1')
    plt.ylabel('Dimension 2')
    plt.legend()
    plt.grid(True)
    plt.show()

# Plot for Buggy PCA
plot_reconstruction(data2D, X_rec_buggy, 'Buggy PCA')

# Plot for Demeaned PCA
plot_reconstruction(data2D, X_rec_demeaned, 'Demeaned PCA')

# Plot for Normalized PCA
plot_reconstruction(data2D, X_rec_normalized, 'Normalized PCA')

# Plot for DRO
plot_reconstruction(data2D, X_rec_dro, 'DRO')
```

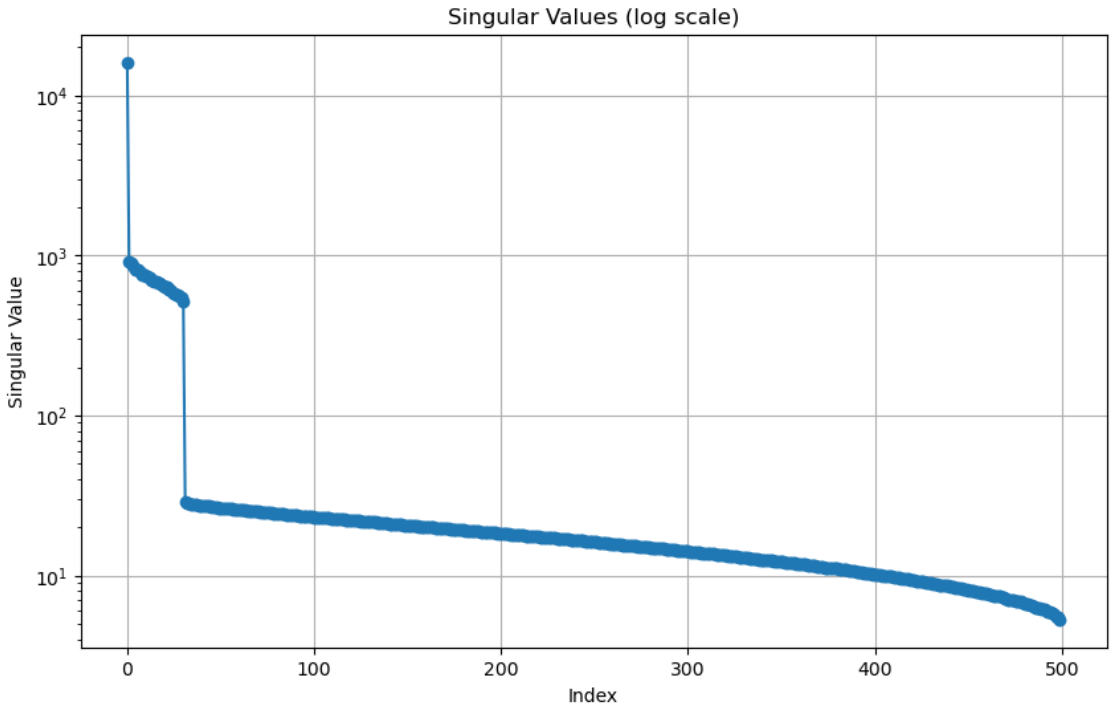



Demeaned PCA Reconstruction
Original Data
Reconstructed Data
Dimension 2
Dimension 1
6.0
5.8
5.6
5.4
5.2
5.0
4
5
6
7
8
9


Normalized PCA Reconstruction
Original Data
Reconstructed Data
Dimension 2
Dimension 1
6.0
5.8
5.6
5.4
5.2
5.0
4
5
6
7
8
9



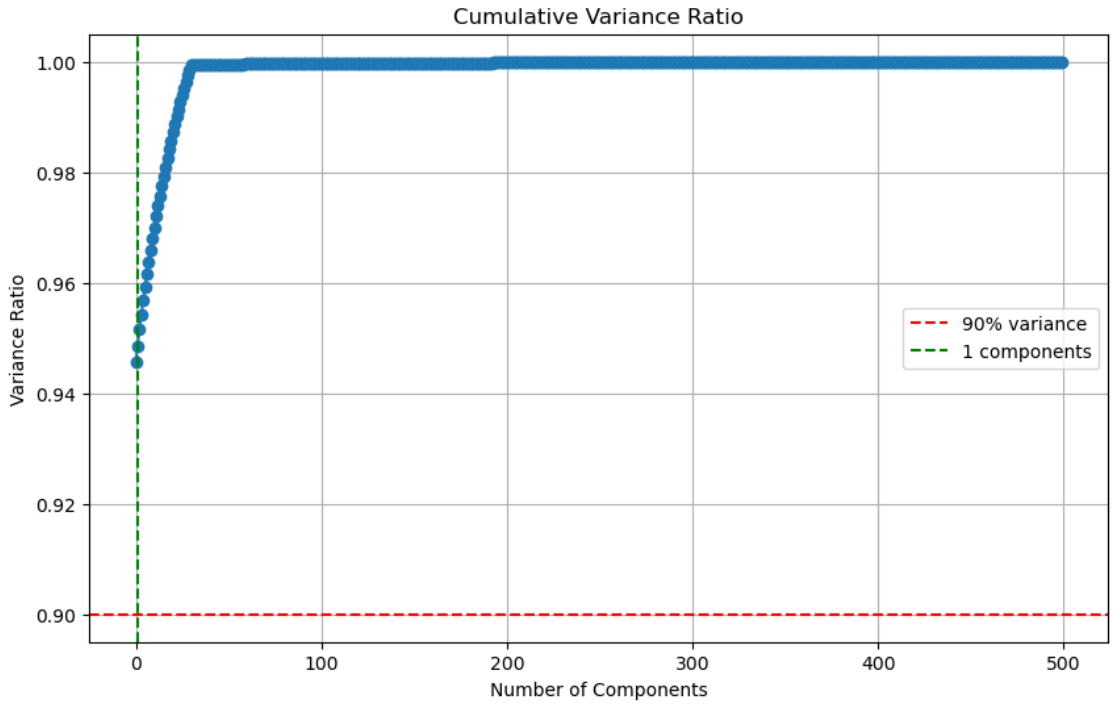
[ ]:
```
# Load the 1000D dataset
data1000D = pd.read_csv("data1000D.csv", header=None).values

# Perform SVD to observe the singular values
U, s, Vt = np.linalg.svd(data1000D, full_matrices=False)

# Plot the singular values
plt.figure(figsize=(10, 6))
plt.plot(s, marker='o')
plt.yscale('log')  # Use logarithmic scale to better visualize the knee point
plt.title('Singular Values (log scale)')
plt.xlabel('Index')
plt.ylabel('Singular Value')
plt.grid(True)
plt.show()

# Return a few singular values for inspection
s[:10]  # Show the first 10 singular values
```



[ ]:
```
array([16045.54068641,   906.79511557,   899.23721784,   856.01200265,
         820.96944086,   819.63910179,   809.67102349,   777.9440578 ,
         754.20297724,   751.87289419])
```

[ ]:
```
# Calculate the cumulative variance explained by the singular values
cumulative_variance = np.cumsum(s**2)
total_variance = cumulative_variance[-1]
variance_ratio = cumulative_variance / total_variance

# Find the number of components needed to capture 90% of the variance
d_90_percent = np.argmax(variance_ratio >= 0.90) + 1  # +1 because index starts␣
 ↪at 0

# Plot the cumulative variance ratio
plt.figure(figsize=(10, 6))
plt.plot(variance_ratio, marker='o')
plt.axhline(y=0.90, color='r', linestyle='--', label='90% variance')
plt.axvline(x=d_90_percent, color='g', linestyle='--', label=f'{d_90_percent}␣
 ↪components')
plt.title('Cumulative Variance Ratio')
plt.xlabel('Number of Components')
plt.ylabel('Variance Ratio')
plt.legend()
plt.grid(True)
```

```
plt.show()

d_90_percent, variance_ratio[:d_90_percent]  # Return the chosen d and variance␣
↪ratios up to d
```



[ ]:
```
(1, array([0.94557254]))
```

The analysis shows that just one component (dimension) captures approximately 94.56% of the total variance, which is more than the 90% threshold commonly used for deciding the number of dimensions. Therefore, for the data1000D dataset, we can reduce the dimensionality from 1000 to 1 without losing much information.

[ ]:
```
# Apply each method to the 1000D dataset with d=1
Z_buggy, V_buggy, X_rec_buggy = buggy_pca(data1000D, d=1)
Z_demeaned, V_demeaned, X_rec_demeaned = demeaned_pca(data1000D, d=1)
Z_normalized, V_normalized, X_rec_normalized = normalized_pca(data1000D, d=1)
Z_dro, A_dro, median_dro, X_rec_dro, error_dro_precomputed =␣
↪robust_dro(data1000D, d=1)

# Calculate reconstruction errors for the data1000D dataset
error_buggy_1000D = reconstruction_error(data1000D, X_rec_buggy)
error_demeaned_1000D = reconstruction_error(data1000D, X_rec_demeaned)
error_normalized_1000D = reconstruction_error(data1000D, X_rec_normalized)
error_dro_1000D = reconstruction_error(data1000D, X_rec_dro)
```

```
# Organize the errors into a DataFrame for better readability
errors_df = pd.DataFrame({
    'Buggy PCA': error_buggy_1000D,
    'Demeaned PCA': error_demeaned_1000D,
    'Normalized PCA': error_normalized_1000D,
    'Robust DRO': error_dro_1000D
}, index=['MSE', 'RMSE', 'Normalized Error'])

errors_df
```

[ ]:

	Buggy PCA	Demeaned PCA	Normalized PCA	Robust DRO
MSE	29.638892	28.021361	28.039611	28.176330
RMSE	5.444161	5.293521	5.295244	5.308138
Normalized Error	0.260546	0.246327	0.246488	0.247689

1. Look at the results for Buggy PCA. The reconstruction error is bad and the reconstructed points don't seem to well represent the original points. Why is this? Hint: Which subspace is Buggy PCA trying to project the points onto?

For Buggy PCA, the reconstruction error is high and the reconstructed points don't accurately represent the original points because this implementation does not center the data by subtracting the mean before performing PCA. PCA is designed to project the data onto the subspace spanned by the eigenvectors (principal components) corresponding to the largest eigenvalues of the covariance matrix of the data.

Without centering, Buggy PCA is effectively finding the principal components of the raw data matrix, not the covariance matrix. This means it projects the points onto a subspace that does not account for the variation about the mean, but rather the variation from the origin, which is not the intended design of PCA. This often results in the first principal component being aligned with the mean of the data instead of the direction of maximum variance, leading to poor reconstruction when projecting back to the original space.

2. The error criterion we are using is the average squared error between the original points and the reconstructed points. In both examples DRO and demeaned PCA achieves the lowest error among all methods. Is this surprising? Why?

It is not surprising that Demeaned PCA achieves the lowest error among all methods for both examples, because demeaned PCA removes the mean, focusing on the variance, which is key for PCA, while Robust DRO is designed to be insensitive to outliers, preserving the structure of the majority of the data. Their effectiveness in reducing error reflects their ability to capture the essential structure of the dataset while reducing dimensionality.