Homework 4

Manyara Bonface Baraka - mbaraka

April 17, 2025

1 Distributed SGD

SGD is an optimization method for training ml models by taking steps in the direction of the negative gradient using samples of the data. In distributed SGD, instead of doing gradient computations on one machine:

- We have **m worker nodes** that each of them compute gradient based on the local data.
- The gradients from all worker nodes are aggregated to update the model parameters.
- This process is repeated iteratively until convergence.

There are two major types of SGD strategies:

1. **Synchronous SGD:** In this strategy, all worker nodes compute their gradients and send them to a central parameter server. The server aggregates the gradients and updates the model parameters. The updated parameters are then sent back to all worker nodes. This process ensures consistency but may suffer from delays caused by slower workers (stragglers).

   Time per iteration = max of all X_i

2. **Asynchronous SGD:** In this strategy, worker nodes compute gradients and send them to the parameter server independently. The server updates the model parameters as soon as it receives gradients from any worker. This approach is faster but may lead to stale updates, as some workers might use outdated model parameters.

   Time per iteration = min of all X_i

Consider that we have a system of m worker nodes and a parameter server performing distributed SGD (stochastic gradient descent). In each iteration, every worker node receives the model from the parameter server, computes one gradient step of the objective function locally using its local data, and sends the gradient to the parameter server. The parameter server does the aggregation of gradients using either synchronous SGD or asynchronous SGD.

The gradient calculation time X_i taken by each node i follows the exponential distribution with rate $\lambda = 2$, which has the following probability density function (PDF):

$$f_X(x) = \begin{cases} \lambda e^{-\lambda x}, & \text{if } x \geq 0 \\ 0, & \text{else} \end{cases}$$

(a) **Cumulative Distribution Function (CDF)**: What is the cumulative distribution function (CDF) of $f_X(x)$, i.e., $F_X(x)$?

Solution

We are given that $X \sim \text{Exponential}(\lambda = 2)$ and the pdf is:

$$f_X(x) \begin{cases} 2e^{-2x}, & \text{if } x \geq 0 \\ 0, & \text{else} \end{cases}$$

The cumulative distribution function (CDF) is the integral of the PDF. For $x \geq 0$, we compute:

$$F_X(x) = \int_0^x f_X(t)\, dt = \int_0^x 2e^{-2t}\, dt$$

Intergration:

$$F_X(x) = \left[-e^{-2t}\right]_0^x = -e^{-2x} + e^0 = 1 - e^{-2x}, \quad \text{for } x \geq 0$$

For $x < 0$, $F_X(x) = 0$ since the PDF is 0 for $x < 0$.

Therefor, the CDF is:

$$F_X(x) = \begin{cases} 1 - e^{-2x}, & \text{if } x \geq 0 \\ 0, & \text{if } x < 0 \end{cases}$$

(b) **Maximum of m Instances**: Define $X_{m:m}$ as the maximum of m i.i.d. (independently and identically distributed) instances $X_1, \ldots, X_m$ following the distribution X. What is the CDF of $X_{m:m}$, and what is the expected value $\mathbb{E}[X_{m:m}]$?

Solution

CDF of the Maximum $X_{m:m}$

The CDF of the maximum $X_{m:m}$ can be derived as follows:

Let $F_X(x)$ be the CDF of a single instance X. The CDF of the maximum $X_{m:m}$ is given by:

$$F_{X_{m:m}}(x) = P(X_{m:m} \leq x) = P(X_1 \leq x, X_2 \leq x, \ldots, X_m \leq x)$$

Since $X_1, X_2, \ldots, X_m$ are i.i.d., we can write:

$$F_{X_{m:m}}(x) = \prod_{i=1}^{m} P(X_i \leq x) = [F_X(x)]^m$$

Substituting the CDF $F_X(x)$ from part (a):

$$F_{X_{m:m}}(x) = \begin{cases} \left(1 - e^{-2x}\right)^m, & \text{if } x \geq 0 \\ 0, & \text{if } x < 0 \end{cases}$$

Expected Value of $X_{m:m}$

The expected value $\mathbb{E}[X_{m:m}]$ of m i.i.d with rate λ is:

$$\mathbb{E}[X_{m:m}] = \frac{1}{\lambda} \sum_{i=1}^{m} \frac{1}{i}$$

Since $\lambda = 2$, therefore:

$$\mathbb{E}[X_{m:m}] = \frac{1}{2} \sum_{i=1}^{m} \frac{1}{i}$$

(c) **Minimum of m Instances**: Define $X_{1:m}$ as the minimum of m i.i.d. instances $X_1, \ldots, X_m$ following the distribution X. What is the CDF of $X_{1:m}$, and what is the expected value $\mathbb{E}[X_{1:m}]$?

Solution

CDF of minimum $X_{1:m}$

The CDF of the minimum $X_{1:m}$ is derived by:

If $F_X(x)$ is the CDF of a single instance X. The CDF of the minimum $X_{1:m}$ is:

$$F_{X_{1:m}}(x) = P(X_{1:m} \leq x) = 1 - P(X_{1:m} > x) = 1 - P(X_1 > x, X_2 > x, \ldots, X_m > x)$$

Since $X_1, X_2, \ldots, X_m$ are i.i.d., we write:

$$F_{X_{1:m}}(x) = 1 - \prod_{i=1}^{m} P(X_i > x) = 1 - [1 - F_X(x)]^m$$

Substituting the CDF $F_X(x)$ from part (a):

$$F_{X_{1:m}}(x) = 1 - e^{-2mx}$$

$$F_{X_{1:m}}(x) = \begin{cases} 1 - e^{-2mx}, & \text{if } x \geq 0 \\ 0, & \text{if } x < 0 \end{cases}$$

Expected Value of $X_{1:m}$

The expected value $\mathbb{E}[X_{1:m}]$ of the minimum of m i.i.d. exponential random variables with rate λ is:

$$\mathbb{E}[X_{1:m}] = \frac{1}{m\lambda}$$

Since $\lambda = 2$, we have:

$$\mathbb{E}[X_{1:m}] = \frac{1}{2m}$$

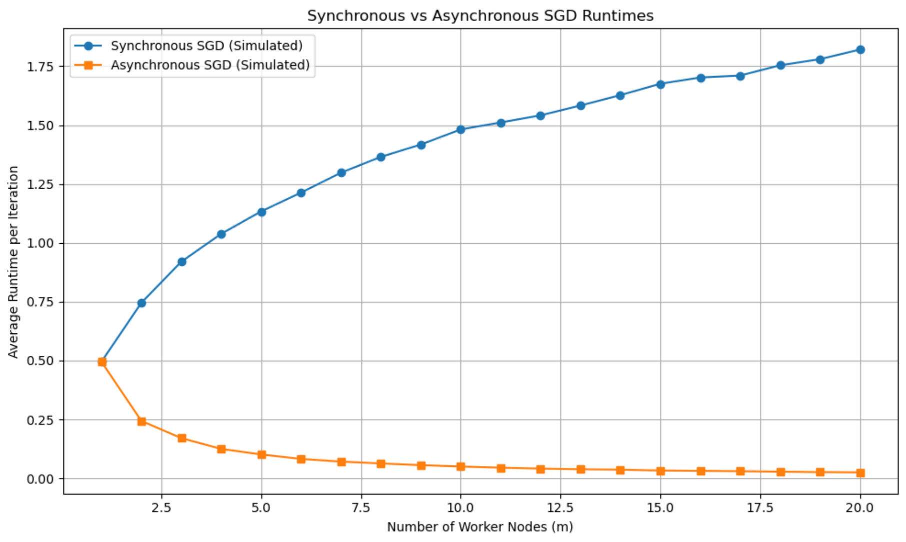
(d) **Simulation of Expected Runtime**: Simulate and compare the expected runtime per iteration of synchronous SGD and asynchronous SGD for different values of m. The time for each worker node to finish one gradient computation is exponentially distributed as given in part (a) with $\lambda = 2$, and it is i.i.d. across workers and iterations. Assume there is no communication delay.

Simulate 5000 iterations of training using Python for different values of m ranging from 1 to 20, and obtain the average runtime per iteration. Make a comparative plot of the average runtimes per iteration of synchronous and asynchronous SGD versus m. Explain the trends observed in the plot in 1-2 sentences. You may use packages inside `numpy.random` to draw random samples from the exponential distribution. Attach your plot and code in PDF format to the end of your homework.

Solution

```
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(42)

lambda_rate = 2
num_iter = 5000
m_values = range(1, 21)

sync_runtime = []
async_runtime = []

for m in m_values:
    # Simulate gradient times: shape (iterations, workers)
    sample = np.random.exponential(scale=1/lambda_rate, size=(num_iter, m))

    # Synchronous: wait for the slowest (max)
    sync_average = np.mean(np.max(sample, axis=1))
    sync_runtime.append(sync_average)

    # Asynchronous: only wait for the fastest (min)
    async_average = np.mean(np.min(sample, axis=1))
    async_runtime.append(async_average)

# Plotting: Simulated Runtimes
plt.figure(figsize=(10, 6))
plt.plot(m_values, sync_runtime, label='Synchronous SGD (Simulated)', marker='o')
plt.plot(m_values, async_runtime, label='Asynchronous SGD (Simulated)', marker='s')
plt.xlabel('Number of Worker Nodes (m)')
plt.ylabel('Average Runtime per Iteration')
plt.title('Synchronous vs Asynchronous SGD Runtimes')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

Output



Trend Explained

- The Sychronous SGD takes longer as m the number of workers increases because it waits for the slowest worker this leads to the max xomputation time grow with number of workers,
- The Assynchoronus SGD stabilizes with more workers this is beacuse the average of exponentially distributed varaibles converges quickly it becomes more efficeint as the number of workers increases.

(e) **Theoretical Expressions for Expected Runtimes**: Write down the theoretical expressions for the expected runtimes per iteration of synchronous and asynchronous SGD in terms of m and λ. (Hint: You can use the expressions derived in parts (b) and (c)). On the figure generated in part (d), also plot the theoretical expected runtimes versus m. Check whether the theoretical and simulated values align.

Solution

Theoretical Expression for Synchronous
From part (b)

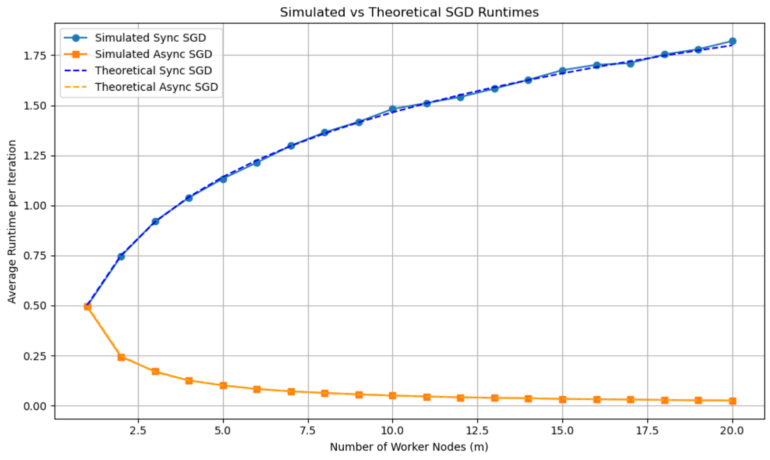
$$\mathbb{E}[X_{m:m}] = \frac{1}{\lambda}\sum_{i=1}^{m}\frac{1}{i}$$

Theoretical Expression for Assynchoronus
Sincew the average of i.i.d exponential random variable with rate λ has

the same mean therefore:

$$\mathbb{E}[X_{1:m}] = \frac{1}{m\lambda}$$

```
# Theoretical Runtimes
harmonic_numbers = np.array([np.sum(1 / np.arange(1, m + 1)) for m in m_values])
expected_sync = (1 / lambda_rate) * harmonic_numbers
expected_async = 1 / (lambda_rate * np.array(list(m_values)))

# Plot: Simulated vs Theoretical
plt.figure(figsize=(10, 6))
plt.plot(m_values, sync_runtime, label='Simulated Sync SGD', marker='o')
plt.plot(m_values, async_runtime, label='Simulated Async SGD', marker='s')
plt.plot(m_values, expected_sync, label='Theoretical Sync SGD', linestyle='--', color='
plt.plot(m_values, expected_async, label='Theoretical Async SGD', linestyle='--', color
plt.xlabel('Number of Worker Nodes (m)')
plt.ylabel('Average Runtime per Iteration')
plt.title('Simulated vs Theoretical SGD Runtimes')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```



Outcome Explained
The theoretical curves follows the simulated curves.

2 K-means

K-means clustering is an unsupervised learning aligorithm that partititions a set of N datapoints to K clusters with an aim of minimizing the within-cluster variance or distortion by assigning each data poitn to a cluster so that the sum of the squared distance between data points and theri correspondin clusters is minimized.

Given a set of data points $\{x_n\}_{n=1}^N$, k-means clustering minimizes the following distortion measure (also called the "objective" or "clustering cost"):

$$D = \sum_{n=1}^{N}\sum_{k=1}^{K} r_{nk}\|x_n - \mu_k\|_2^2$$

where μ_k is the prototype of the k-th cluster and r_{nk} is a binary indicator variable. If x_n is assigned to the cluster k, r_{nk} is 1, and otherwise r_{nk} is 0. For each cluster, μ_k is the prototype representative for all the data points assigned to that cluster.

(a) [10 points] In lecture, we stated but did not prove that μ_k is the mean of all points associated with the k-th cluster, thus motivating the name of the algorithm. You will now prove this statement. Assuming all r_{nk} are known (i.e., assuming you know the cluster assignments of all N data points), show that the objective D is minimized when each μ_k is chosen as the mean of all data points assigned to cluster k, for any k. This justifies the iterative procedure of k-means.

Solution

To minimize the objective D, we differentiate it with respect to μ_k and set the derivative to zero. The objective D is given by:

$$D = \sum_{n=1}^{N}\sum_{k=1}^{K} r_{nk}\|x_n - \mu_k\|_2^2$$

Expanding the squared norm:

$$\|x_n - \mu_k\|_2^2 = (x_n - \mu_k)^\top (x_n - \mu_k) = x_n^\top x_n - 2x_n^\top \mu_k + \mu_k^\top \mu_k$$

Substituting this into D:

$$D = \sum_{n=1}^{N}\sum_{k=1}^{K} r_{nk}\left(x_n^\top x_n - 2x_n^\top \mu_k + \mu_k^\top \mu_k\right)$$

Rearranging terms:

$$D = \sum_{n=1}^{N}\sum_{k=1}^{K} r_{nk} x_n^\top x_n - 2\sum_{n=1}^{N}\sum_{k=1}^{K} r_{nk} x_n^\top \mu_k + \sum_{n=1}^{N}\sum_{k=1}^{K} r_{nk} \mu_k^\top \mu_k$$

The first term does not depend on μ_k, so we can ignore it when minimizing D. The remaining terms are:

$$D = -2\sum_{n=1}^{N}\sum_{k=1}^{K} r_{nk} x_n^\top \mu_k + \sum_{n=1}^{N}\sum_{k=1}^{K} r_{nk} \mu_k^\top \mu_k$$

Now, differentiate D with respect to μ_k:

$$\frac{\partial D}{\partial \mu_k} = -2\sum_{n=1}^{N} r_{nk} x_n + 2\mu_k \sum_{n=1}^{N} r_{nk}$$

Set $\frac{\partial D}{\partial \mu_k} = 0$ to find the optimal μ_k:

$$-2\sum_{n=1}^{N} r_{nk} x_n + 2\mu_k \sum_{n=1}^{N} r_{nk} = 0$$

Simplify:

$$\mu_k \sum_{n=1}^{N} r_{nk} = \sum_{n=1}^{N} r_{nk} x_n$$

Solve for μ_k:

$$\mu_k = \frac{\sum_{n=1}^{N} r_{nk} x_n}{\sum_{n=1}^{N} r_{nk}}$$

This shows that μ_k is the mean of all points assigned to cluster k, as required.

(b) [10 points] As discussed in lecture, sometimes we wish to scale each feature in order to ensure that "larger" features do not dominate the clustering. Suppose that each data point x_n is a d-dimensional feature vector and that we scale the j-th feature by a factor $w_j > 0$. Letting W denote a $d \times d$ diagonal matrix with the j-th diagonal entry being w_j, $j = 1, 2, \ldots, d$, we can write our transformed features as $x' = Wx$.

Suppose we fix the r_{nk}, i.e., we take the assignment of data points x_n to clusters k as given. Our goal is then to find the cluster centers μ_k that minimize the distortion measure:

$$D = \sum_{n=1}^{N}\sum_{k=1}^{K} r_{nk}\|Wx_n - \mu_k\|_2^2.$$

Show that the cluster centers $\{\mu_k\}$ that do so are given by:

$$\mu_k = \left(\sum_{n=1}^{N} r_{nk}\right)^{-1} W \sum_{n=1}^{N} r_{nk}x_n.$$

Solution

To minimize the distortion D, we differentiate it with respect to μ_k and set the derivative to zero. The distortion D is given by:

$$D = \sum_{n=1}^{N}\sum_{k=1}^{K} r_{nk}\|Wx_n - \mu_k\|_2^2$$

Expanding the squared norm:

$$\|Wx_n-\mu_k\|_2^2 = (Wx_n-\mu_k)^\top(Wx_n-\mu_k) = x_n^\top W^\top Wx_n - 2x_n^\top W^\top \mu_k + \mu_k^\top \mu_k$$

Substituting this into D:

$$D = \sum_{n=1}^{N}\sum_{k=1}^{K} r_{nk}\left(x_n^\top W^\top Wx_n - 2x_n^\top W^\top \mu_k + \mu_k^\top \mu_k\right)$$

Rearranging terms:

$$D = \sum_{n=1}^{N}\sum_{k=1}^{K} r_{nk}x_n^\top W^\top Wx_n - 2\sum_{n=1}^{N}\sum_{k=1}^{K} r_{nk}x_n^\top W^\top \mu_k + \sum_{n=1}^{N}\sum_{k=1}^{K} r_{nk}\mu_k^\top \mu_k$$

The first term does not depend on μ_k, so we can ignore it when minimizing D. The remaining terms are:

$$D = -2\sum_{n=1}^{N}\sum_{k=1}^{K} r_{nk}x_n^\top W^\top \mu_k + \sum_{n=1}^{N}\sum_{k=1}^{K} r_{nk}\mu_k^\top \mu_k$$

Now, differentiate D with respect to μ_k:

$$\frac{\partial D}{\partial \mu_k} = -2 \sum_{n=1}^{N} r_{nk} W x_n + 2\mu_k \sum_{n=1}^{N} r_{nk}$$

Set $\frac{\partial D}{\partial \mu_k} = 0$ to find the optimal μ_k:

$$-2 \sum_{n=1}^{N} r_{nk} W x_n + 2\mu_k \sum_{n=1}^{N} r_{nk} = 0$$

Simplify:

$$\mu_k \sum_{n=1}^{N} r_{nk} = \sum_{n=1}^{N} r_{nk} W x_n$$

Solve for μ_k:

$$\mu_k = \left(\sum_{n=1}^{N} r_{nk} \right)^{-1} \sum_{n=1}^{N} r_{nk} W x_n$$

Thus, the cluster centers μ_k are given by:

$$\mu_k = \frac{\sum_{n=1}^{N} r_{nk} W x_n}{\sum_{n=1}^{N} r_{nk}}$$

3 3-Dimensional Principal Component Analysis

In this problem, we will perform PCA on 3-dimensional data step by step. We are given three data points:

$$x_1 = [0, -1, -2], \quad x_2 = [1, 1, 1], \quad x_3 = [2, 0, 1],$$

and we want to find 2 principal components of the given data.

(a) [8 points] First, find the covariance matrix $C_X = X^T X$ where:

$$X = \begin{bmatrix} x_1 - \bar{x} \\ x_2 - \bar{x} \\ x_3 - \bar{x} \end{bmatrix},$$

and $\bar{x} = \frac{1}{3}(x_1 + x_2 + x_3)$ is the mean of the data samples. Then, find the eigenvalues and the corresponding eigenvectors of C_X. Feel free to use any numerical analysis program such as `numpy`, e.g., `numpy.linalg.eig` can be useful. However, you should explain what you inputted into this program.

Solution

First, calculate the mean of the data samples:

$$\bar{x} = \frac{1}{3}(x_1 + x_2 + x_3) = \frac{1}{3}\left([0, -1, -2] + [1, 1, 1] + [2, 0, 1]\right) = [1, 0, 0].$$

Next, subtract the mean from each data point to form the matrix X:

$$X = \begin{bmatrix} x_1 - \bar{x} \\ x_2 - \bar{x} \\ x_3 - \bar{x} \end{bmatrix} = \begin{bmatrix} [0, -1, -2] - [1, 0, 0] \\ [1, 1, 1] - [1, 0, 0] \\ [2, 0, 1] - [1, 0, 0] \end{bmatrix} = \begin{bmatrix} -1 & -1 & -2 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}.$$

The covariance matrix C_X is given by:

$$C_X = X^\top X.$$

Compute $X^\top$:

$$X^\top = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 1 & 0 \\ -2 & 1 & 1 \end{bmatrix}.$$

Now, calculate C_X:

$$C_X = X^\top X = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 1 & 0 \\ -2 & 1 & 1 \end{bmatrix} \begin{bmatrix} -1 & -1 & -2 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}.$$

Matrix multiplication:

$$C_X = \begin{bmatrix} 2 & 1 & 3 \\ 1 & 2 & 3 \\ 3 & 3 & 6 \end{bmatrix}.$$

Found the Eigenvalues and eigenvectors:

```
import numpy as np

Cx = np.array([
    [2, 1, 3],
    [1, 2, 3],
    [3, 3, 6]
])

eigvals, eigvecs = np.linalg.eig(Cx)
print("EigenValues:", eigvals)
print("Eigen Vectors", eigvecs)
```

Eigenvalues:

$$\lambda_1 = 9.000, \quad \lambda_2 = 1.000, \quad \lambda_3 = 1.054 \times 10^{-16}.$$

Eigenvectors (corresponding to $\lambda_1, \lambda_2, \lambda_3$):

$$u_1 = \begin{bmatrix} -0.408 \\ -0.408 \\ -0.816 \end{bmatrix}, \quad u_2 = \begin{bmatrix} -0.707 \\ 0.707 \\ 0.000 \end{bmatrix}, \quad u_3 = \begin{bmatrix} -0.577 \\ -0.577 \\ 0.577 \end{bmatrix}.$$

(b) [4 points] Using the result above, find the first two principal components of the given data.

Solution

From part a above, to find the first top 2 princible component we select the top 2 eigenvectors corresponding to the top eigenvalues.

$$\lambda_1 = 9.000, \quad \lambda_2 = 1.000,$$

The eigenvectors corresponding to λ_1 and λ_2 are:

$$u_1 = \begin{bmatrix} -0.408 \\ -0.408 \\ -0.816 \end{bmatrix}, \quad u_2 = \begin{bmatrix} -0.707 \\ 0.707 \\ 0.000 \end{bmatrix}.$$

Therefore, the first two principal components are:

$$u_1 = \begin{bmatrix} -0.408 \\ -0.408 \\ -0.816 \end{bmatrix}, \quad u_2 = \begin{bmatrix} -0.707 \\ 0.707 \\ 0.000 \end{bmatrix}.$$

(c) [8 points] Now we want to represent the data $x_1, \cdots, x_3$ using a 2-dimensional subspace instead of a 3-dimensional one. PCA gives us the 2-D plane which minimizes the difference between the original data and the data projected to the 2-dimensional plane. In other words, x_i can be approximated as:

$$\hat{x}_i = a_{i1}u_1 + a_{i2}u_2 + \bar{x},$$

where u_1 and u_2 are the principal components we found in 3.(b). Figure 1 gives an example of what this might look like.

Find a_{i1}, a_{i2} for $i = 1, 2, 3$. Then, find the $\hat{x}_i$'s and the difference between $\hat{x}_i$ and x_i, i.e., $\|\hat{x}_i - x_i\|_2$ for $i = 1, 2, 3$. (Again, feel free to use any numerical analysis program to get the final answer, but show your calculation process.)

Solution

Have defined the calculation steps in the comments.

```
import numpy as np

# Original data
data = np.array([
    [0, -1, -2],
    [1,  1,  1],
    [2,  0,  1]
])

# Compute the mean
mean_vector = np.mean(data, axis=0)

# Define the Center the data
X_centered = data - mean_vector

# Given covariance matrix
```

```
Cx = np.array([
    [2, 1, 3],
    [1, 2, 3],
    [3, 3, 6]
])

# Eigen decomposition
eigvals, eigvecs = np.linalg.eig(Cx)
print("Eigenvalues:", eigvals)
print("Eigenvectors:\n", eigvecs)

# Furst 2 principal components (eigenvectors)
U = eigvecs[:, :2]  # Take first two columns (PC1 and PC2)

# Project centered data onto 2D principal subspace
A = X_centered @ U

# Reconstructing the data using the top 2 PCs
X_reconstructed = A @ U.T + mean_vector

# Computing reconstruction errors
errors = np.sum((data - X_reconstructed)**2, axis=1)

# Display results
print("\nProjection coefficients (a_i):\n", A)
print("\nReconstructed data (x_i_hat):\n", X_reconstructed)
print("\nReconstruction errors ||x_i - x_i_hat||^2:\n", errors)
```

Output: Projection coefficients (a_i):

$$\begin{bmatrix} 2.44948974 \times 10^{0} & 2.62480955 \times 10^{-17} \\ -1.22474487 \times 10^{0} & 7.07106781 \times 10^{-1} \\ -1.22474487 \times 10^{0} & -7.07106781 \times 10^{-1} \end{bmatrix}$$

Reconstructed data ($\hat{x}_i$):

$$\begin{bmatrix} 0.00000000 \times 10^{0} & -1.00000000 \times 10^{0} & -2.00000000 \times 10^{0} \\ 1.00000000 \times 10^{0} & 1.00000000 \times 10^{0} & 1.00000000 \times 10^{0} \\ 2.00000000 \times 10^{0} & 1.66533454 \times 10^{-16} & 1.00000000 \times 10^{0} \end{bmatrix}$$

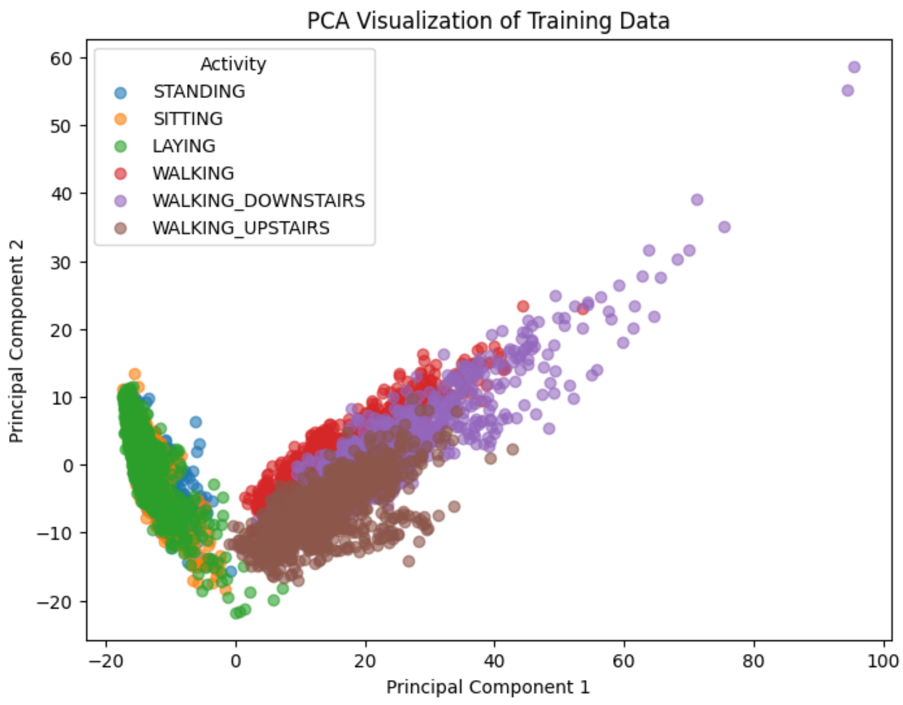
Reconstruction errors $\|\hat{x}_i - x_i\|_2^2$:

$$\begin{bmatrix} 1.97215226 \times 10^{-31} \\ 2.58844985 \times 10^{-31} \\ 2.77333912 \times 10^{-32} \end{bmatrix}$$

4 Clustering Human Activity using Inertial Sensors Data

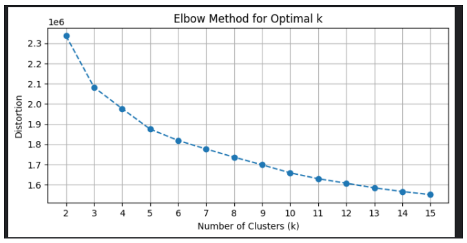
4.1 Import Data and Plotting

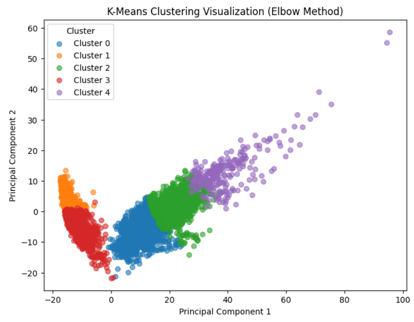
Visual of PCA wiht different colors with respective activity labels



4.2 Choosing the optimal Number of Clusters

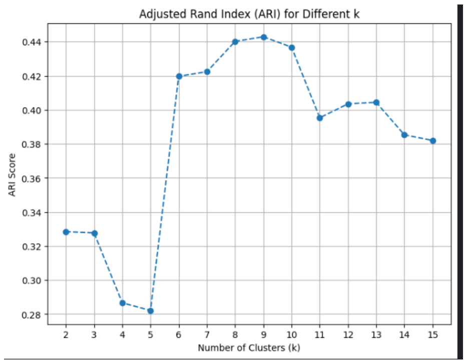
(a) **Elbow Method**

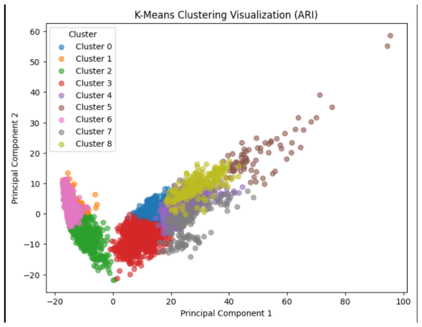




- The optimal number of cluser is 5.
- Distortion measures the sum of the squared distance between the data points and the assigned clustes this means an increase in the clusters leads to the decrease of the the distortion. This is mainly because increaseing clusters allows the centroing to better fit the data which reduces the distance between points and their nearest centroid. However, as K becomes large the distortion the decreasing rate also diminishes which leads to the elbow point where addding more clustersprovides the diminishing returns in reducing distortion.

(b) **Adjusted Rand Index**





- The optimal was 9.
- ARI measures the similarity between clustering results and the ground truth labels. As the number of clusters increase the followng trend is observeds.

- Plateau: Beyond the optimal K, the Ari scrore declines as the clusters become too granular, this leads reduction of alignment with the ground truth.
- Fluctuations: As K increases, the ARI fluctuares due to creation of more clustres that do not correspond well to the true data distributions or overfitting.

4.3 Prototype Selection using K-means clustering

(a) **Random Selection** : Average Accuracy with Random Selection over 10 repetitions: 0.9192.

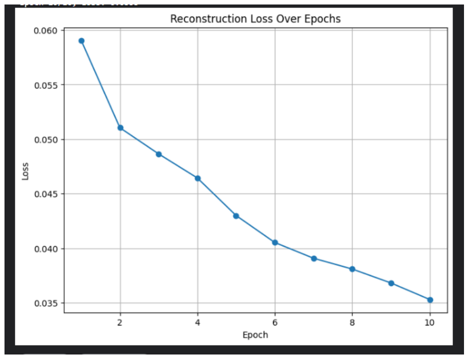
(b) **Using K-means clustering by Class** : Accuracy with K-means Selection: 0.8931.

Comparizon of the Random selection and the K-meands clusering by class model Accuracy

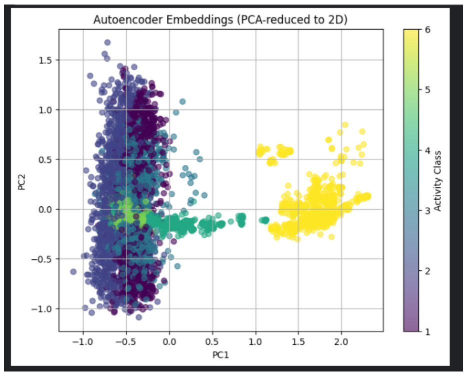
The Random prototype selection had a higher Accuracy of 92.23% compared to the K-means clustering that had 90.57%, this could be becaue Random seelction relies on randomly selected prototypes that may capture diverse sample from each class while K-means clustering selects the prototype based on clusters that ensure that the selected samples are representative of the cluser centroids however, this may miss some outlier or diverse samples. And since the later focuess more on centroirds rather than capturing the full variablity of the data this may be the reason of the less performance of the models compared to the reandom selection model.

4.4 Autoencoder for Feature Learning

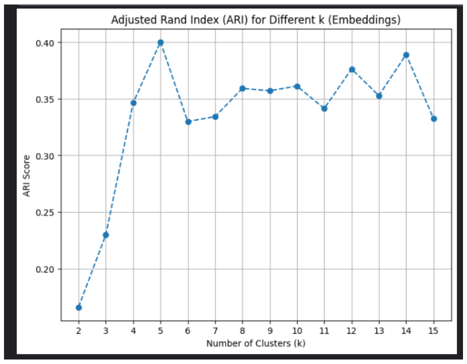
Reconstruction



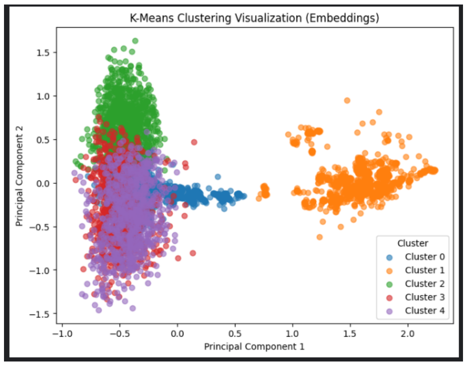
Autoencoder Embedding



ARI for for different k-means clustering



K-means Clustering Visualization



Clustering Human Activity using Inertial Sensors Data

Note:

- Use the next cell to download the data directly, if that didn't work. you can download it manually (available at UCI archive) a copy will also be available on Piazza.
- Don't change the part of the code that labels `#Do not change`
- Attach this notebook to your answer sheet with all outputs visible.
- make sure you have `pytorch, scikit learn, pandas` in your environment

```
#### Download the dataset

import urllib.request
import zipfile
import os

dataset_url =
"https://archive.ics.uci.edu/static/public/240/human+activity+recognit
ion+using+smartphones.zip"
zip_file_path = "Dataset.zip"
extracted_downloaded_folder = "Dataset"
extracted_data_folder = "UCI HAR Dataset"

if not os.path.exists(zip_file_path):
    print("Downloading the dataset...")
    urllib.request.urlretrieve(dataset_url, zip_file_path)

if not os.path.exists(extracted_downloaded_folder):
    print("Extracting the dataset...")
    with zipfile.ZipFile(zip_file_path, 'r') as zip_ref:
        zip_ref.extractall(".")

if not os.path.exists(extracted_data_folder):
    print("Extracting the dataset...")
    with zipfile.ZipFile(extracted_data_folder +'.zip', 'r') as
zip_ref:
        zip_ref.extractall(".")

print("Dataset is ready.")
```

```
Downloading the dataset...
Extracting the dataset...
Extracting the dataset...
Dataset is ready.
```

Load the data into a dataframe

```
# Import necessary libraries
import pandas as pd
import matplotlib.pyplot as plt

# Define paths to data files
train_path = "UCI HAR Dataset/train"  # TODO
test_path = "UCI HAR Dataset/test"   # TODO
activity_mapper_path = "UCI HAR Dataset/activity_labels.txt"


# Load training and testing data
X_train = pd.read_csv(f"{train_path}/X_train.txt",
delim_whitespace=True, header=None)
y_train = pd.read_csv(f"{train_path}/y_train.txt",
delim_whitespace=True, header=None)
X_test, y_test =   pd.read_csv(f"{test_path}/X_test.txt",
delim_whitespace=True, header=None),
pd.read_csv(f"{test_path}/y_test.txt", delim_whitespace=True,
header=None)  # TODO



# Display the first 5 rows of the training dataframe
print("First 5 rows of training feature dataframe:")
X_train.head()  # DO NOT CHANGE
```

```
<ipython-input-5-1e64e906fa49>:12: FutureWarning: The
'delim_whitespace' keyword in pd.read_csv is deprecated and will be
removed in a future version. Use ``sep='\s+'`` instead
  X_train = pd.read_csv(f"{train_path}/X_train.txt",
delim_whitespace=True, header=None)
<ipython-input-5-1e64e906fa49>:13: FutureWarning: The
'delim_whitespace' keyword in pd.read_csv is deprecated and will be
removed in a future version. Use ``sep='\s+'`` instead
  y_train = pd.read_csv(f"{train_path}/y_train.txt",
delim_whitespace=True, header=None)
<ipython-input-5-1e64e906fa49>:14: FutureWarning: The
'delim_whitespace' keyword in pd.read_csv is deprecated and will be
removed in a future version. Use ``sep='\s+'`` instead
  X_test, y_test =   pd.read_csv(f"{test_path}/X_test.txt",
delim_whitespace=True, header=None),
pd.read_csv(f"{test_path}/y_test.txt", delim_whitespace=True,
header=None)  # TODO
```

```
First 5 rows of training feature dataframe:

<ipython-input-5-1e64e906fa49>:14: FutureWarning: The
'delim_whitespace' keyword in pd.read_csv is deprecated and will be
removed in a future version. Use ``sep='\s+'`` instead
  X_test, y_test =   pd.read_csv(f"{test_path}/X_test.txt",
delim_whitespace=True, header=None),
pd.read_csv(f"{test_path}/y_test.txt", delim_whitespace=True,
header=None)  # TODO

        0         1         2         3         4         5         6
\
0  0.288585 -0.020294 -0.132905 -0.995279 -0.983111 -0.913526 -
0.995112
1  0.278419 -0.016411 -0.123520 -0.998245 -0.975300 -0.960322 -
0.998807
2  0.279653 -0.019467 -0.113462 -0.995380 -0.967187 -0.978944 -
0.996520
3  0.279174 -0.026201 -0.123283 -0.996091 -0.983403 -0.990675 -
0.997099
4  0.276629 -0.016570 -0.115362 -0.998139 -0.980817 -0.990482 -
0.998321

        7         8         9    ...       551       552       553
554  \
0 -0.983185 -0.923527 -0.934724  ... -0.074323 -0.298676 -0.710304 -
0.112754
1 -0.974914 -0.957686 -0.943068  ...  0.158075 -0.595051 -0.861499
0.053477
2 -0.963668 -0.977469 -0.938692  ...  0.414503 -0.390748 -0.760104 -
0.118559
3 -0.982750 -0.989302 -0.938692  ...  0.404573 -0.117290 -0.482845 -
0.036788
4 -0.979672 -0.990441 -0.942469  ...  0.087753 -0.351471 -0.699205
0.123320

        555       556       557       558       559       560
0  0.030400 -0.464761 -0.018446 -0.841247  0.179941 -0.058627
1 -0.007435 -0.732626  0.703511 -0.844788  0.180289 -0.054317
2  0.177899  0.100699  0.808529 -0.848933  0.180637 -0.049118
3 -0.012892  0.640011 -0.485366 -0.848649  0.181935 -0.047663
4  0.122542  0.693578 -0.615971 -0.847865  0.185151 -0.043892

[5 rows x 561 columns]
```

scaling the data and PCA

```python
from sklearn.preprocessing import StandardScaler
# TODO: Scale X_train
scaler = StandardScaler()
```

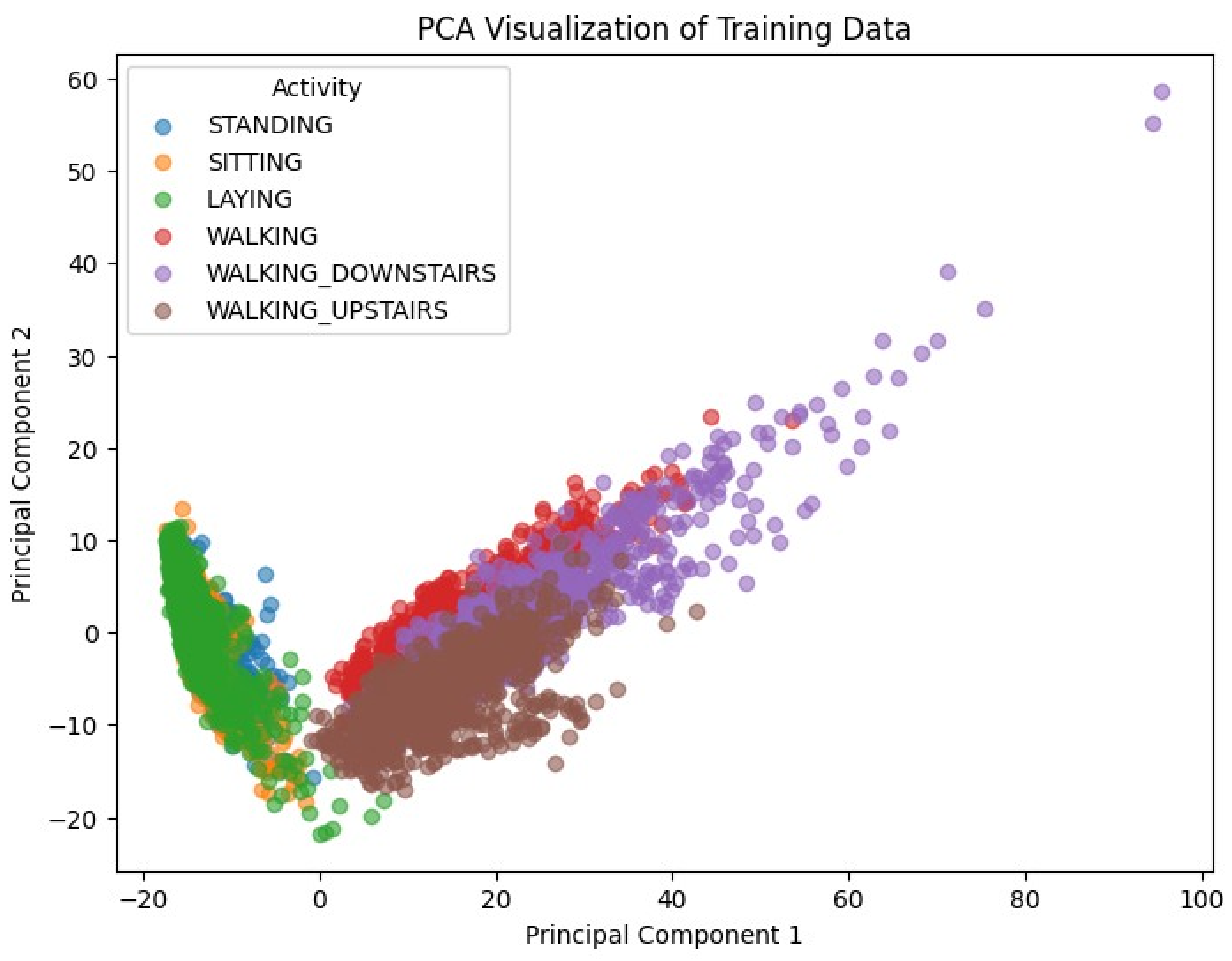
```
X_train_scaled = scaler.fit_transform(X_train)

# TODO: Scale X_test
X_test_scaled = scaler.transform(X_test)

# Convert scaled arrays back to DataFrames
X_train = pd.DataFrame(X_train_scaled)
X_test = pd.DataFrame(X_test_scaled)


# Add 'Activity' column to create training_df and testing_df
# TODO: Combine X_train and y_train into a single DataFrame named
training_df.
training_df = pd.concat([X_train, y_train.rename(columns={0:
'Activity'})], axis=1)

# TODO: Combine X_test and y_test into a single DataFrame named
testing_df.
testing_df = pd.concat([X_test, y_test.rename(columns={0:
'Activity'})], axis=1)



# Display the first 5 rows of the training feature dataframe
print("First 5 rows of training feature dataframe:")
training_df.head()  # DO NOT CHANGE
```

```
First 5 rows of training feature dataframe:

          0         1         2         3         4         5
6  \
0  0.200642 -0.063683 -0.419628 -0.868814 -0.939441 -0.737529 -
0.859817
1  0.055948  0.031486 -0.253908 -0.875426 -0.923902 -0.849304 -
0.868531
2  0.073515 -0.043416 -0.076295 -0.869039 -0.907760 -0.893785 -
0.863137
3  0.066696 -0.208422 -0.249712 -0.870626 -0.940022 -0.921805 -
0.864503
4  0.030469  0.027587 -0.109848 -0.875188 -0.934878 -0.921343 -
0.867384

          7         8         9  ...       552       553       554
555  \
0 -0.939019 -0.766437 -0.856036  ...  0.025960 -0.276399 -0.360603
0.062940
1 -0.921998 -0.848928 -0.871359  ... -0.897357 -0.767990  0.133011 -
0.021461
2 -0.898854 -0.896701 -0.863323  ... -0.260878 -0.438316 -0.377840
0.391976
3 -0.938124 -0.925279 -0.863323  ...  0.591045  0.463155 -0.135025 -
```

```
0.033637
4 -0.931789 -0.928028 -0.870260  ... -0.138515 -0.240313  0.340406
0.268486

        556       557       558       559       560  Activity
0 -0.778427 -0.026080 -0.687219  0.407946 -0.007568         5
1 -1.218805  1.484470 -0.694138  0.409117  0.007875         5
2  0.151207  1.704201 -0.702239  0.410288  0.026502         5
3  1.037851 -1.003019 -0.701684  0.414650  0.031714         5
4  1.125918 -1.276282 -0.700152  0.425463  0.045225         5

[5 rows x 562 columns]

# TODO perform PCA on the train data and get the first 2 PC
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
X_train_pca = pca.fit_transform(X_train)  #TODO
```

Visualize the data

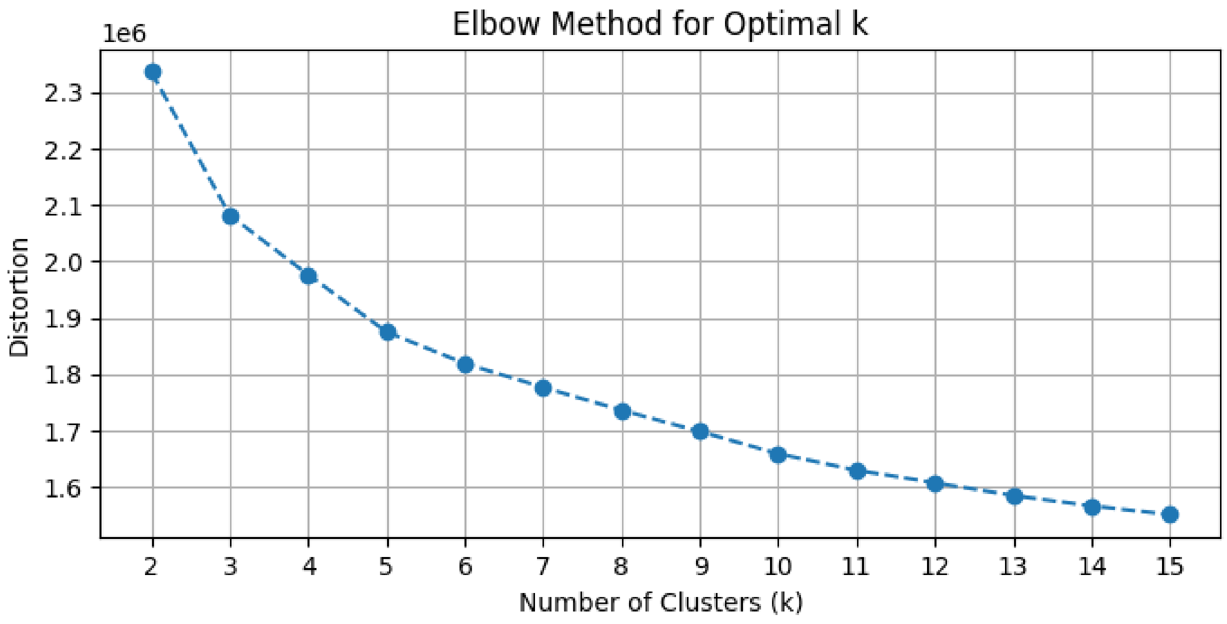
```
# Visualize training data using PCA

# Use the feature decoder to create Activity Name column

# Load activity labels
activity_labels = pd.read_csv(activity_mapper_path, header=None,
sep='\s+', names=['id', 'activity_name'])

# Create mapping dictionary {1: "WALKING", 2: "WALKING_UPSTAIRS", ...}
activity_mapping = dict(zip(activity_labels['id'],
activity_labels['activity_name']))

# Use the mapping to decode the Activities labels
Activity_Name = y_train[0].map(activity_mapping)

# Create a scatter plot using the X_train_pca and the Activity Names
plt.figure(figsize=(8, 6))
for activity in Activity_Name.unique():
    indices = Activity_Name == activity
    plt.scatter(X_train_pca[indices, 0], X_train_pca[indices, 1],
label=activity, alpha=0.6)


# TODO <--code below-->
plt.title('PCA Visualization of Training Data')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.legend(title='Activity')
plt.show()
```



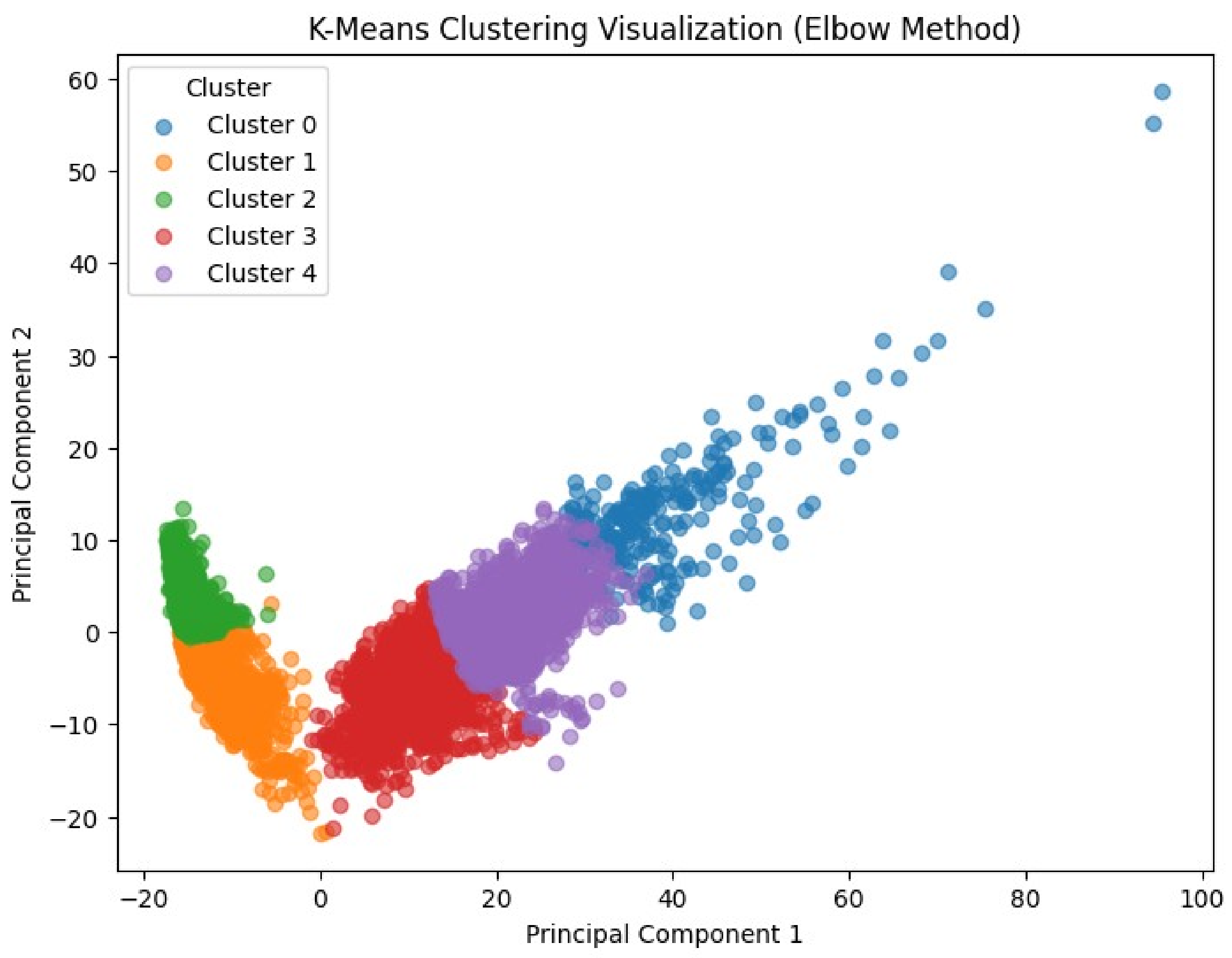
Kmeans Clustering and The Optimal Number of Clusters

1. **Elbow Method**

```
from sklearn.cluster import KMeans

# Elbow Method

distortion_values = []
for k in range(2, 16):
    kmeans = KMeans(n_clusters=k, random_state=42, n_init=10)
    kmeans.fit(X_train_scaled)
    distortion_values.append(kmeans.inertia_)

# Plotting the Elbow Method
plt.figure(figsize=(8, 3.5))
plt.plot(range(2, 16), distortion_values, marker='o', linestyle='--')
plt.title('Elbow Method for Optimal k')
plt.xlabel('Number of Clusters (k)')
plt.ylabel('Distortion')
plt.xticks(range(2, 16))
plt.grid()
plt.show()
```



```
# Choose k based on the elbow method
elbow_k = 5
kmeans_elbow = KMeans(n_clusters=elbow_k, random_state=42, n_init=10)
clusters_elbow = kmeans_elbow.fit_predict(X_train_scaled)

# PCA for visualization - already computed PCA
X_train_pca_elbow = X_train_pca

# Plotting the clusters
plt.figure(figsize=(8, 6))
for cluster in range(elbow_k):
    indices = clusters_elbow == cluster
    plt.scatter(X_train_pca_elbow[indices, 0],
X_train_pca_elbow[indices, 1], label=f'Cluster {cluster}', alpha=0.6)

plt.title('K-Means Clustering Visualization (Elbow Method)')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.legend(title='Cluster')
plt.show()
```



Explain distortion with Increasing K

- The optimal number of cluser is 5.
  - Distortion measures the sum of the squared distance between the data points and the assigned clustes this means an increase in the clusters leads to the decrease of the the distortion. This is mainly because increaseing clusters allows the centroing to better fit the data which reduces the distance between points and their nearest centroid. However, as K becomes large the distortion the decreasing rate also diminishes which leads to the elbow point where addding more clustersprovides the diminishing returns in reducing distortion.

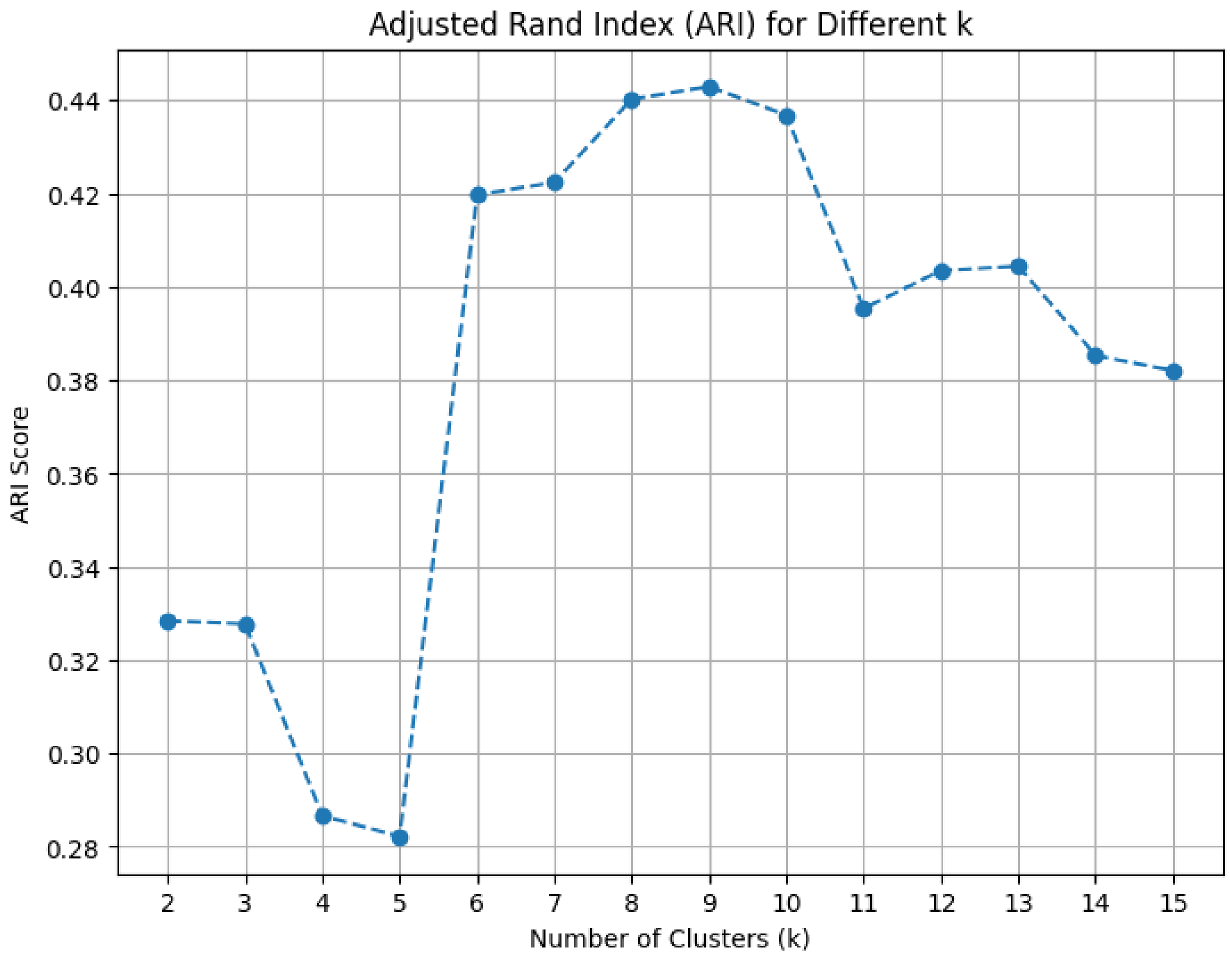
2. Adjusted Rand Index (ARI)

```
from sklearn.metrics import adjusted_rand_score

# 2. Adjusted Rand Index (ARI)

ari_scores = []
for k in range(2, 16):
    kmeans = KMeans(n_clusters=k, random_state=42)
    clusters = kmeans.fit_predict(X_train_scaled)
```

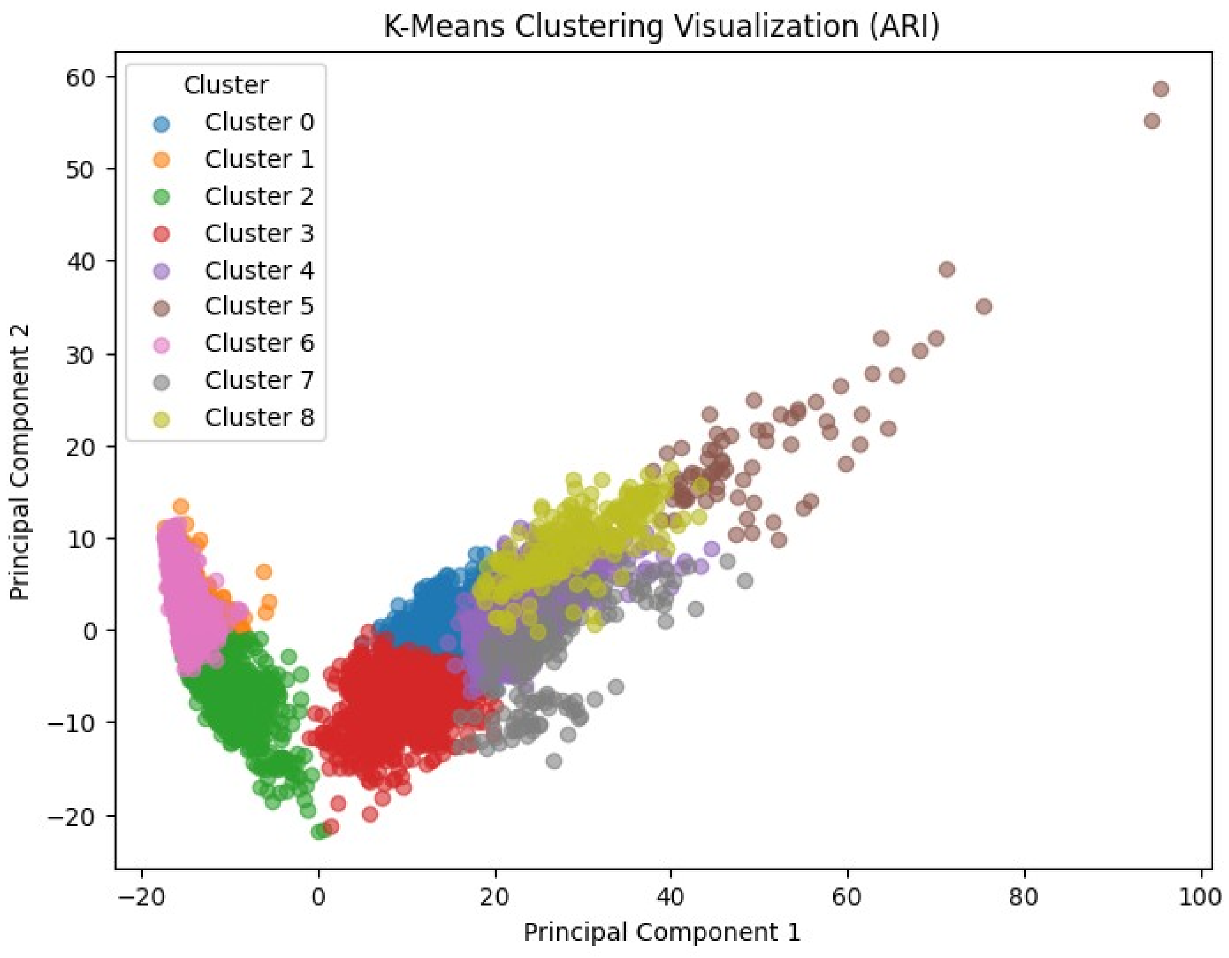
```python
    ari = adjusted_rand_score(y_train[0], clusters)
    ari_scores.append(ari)

# Plotting ARI Scores
plt.figure(figsize=(8, 6))
plt.plot(range(2, 16), ari_scores, marker='o', linestyle='--')
plt.title('Adjusted Rand Index (ARI) for Different k')
plt.xlabel('Number of Clusters (k)')
plt.ylabel('ARI Score')
plt.xticks(range(2, 16))
plt.grid()
plt.show()
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/
_kmeans.py:870: FutureWarning: The default value of `n_init` will
change from 10 to 'auto' in 1.4. Set the value of `n_init` explicitly
to suppress the warning
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870
: FutureWarning: The default value of `n_init` will change from 10 to
'auto' in 1.4. Set the value of `n_init` explicitly to suppress the
warning
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870
: FutureWarning: The default value of `n_init` will change from 10 to
'auto' in 1.4. Set the value of `n_init` explicitly to suppress the
warning
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870
: FutureWarning: The default value of `n_init` will change from 10 to
'auto' in 1.4. Set the value of `n_init` explicitly to suppress the
warning
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870
: FutureWarning: The default value of `n_init` will change from 10 to
'auto' in 1.4. Set the value of `n_init` explicitly to suppress the
warning
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870
: FutureWarning: The default value of `n_init` will change from 10 to
'auto' in 1.4. Set the value of `n_init` explicitly to suppress the
warning
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870
: FutureWarning: The default value of `n_init` will change from 10 to
'auto' in 1.4. Set the value of `n_init` explicitly to suppress the
warning
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870
: FutureWarning: The default value of `n_init` will change from 10 to
```

```
'auto' in 1.4. Set the value of `n_init` explicitly to suppress the
warning
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870
: FutureWarning: The default value of `n_init` will change from 10 to
'auto' in 1.4. Set the value of `n_init` explicitly to suppress the
warning
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870
: FutureWarning: The default value of `n_init` will change from 10 to
'auto' in 1.4. Set the value of `n_init` explicitly to suppress the
warning
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870
: FutureWarning: The default value of `n_init` will change from 10 to
'auto' in 1.4. Set the value of `n_init` explicitly to suppress the
warning
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870
: FutureWarning: The default value of `n_init` will change from 10 to
'auto' in 1.4. Set the value of `n_init` explicitly to suppress the
warning
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870
: FutureWarning: The default value of `n_init` will change from 10 to
'auto' in 1.4. Set the value of `n_init` explicitly to suppress the
warning
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870
: FutureWarning: The default value of `n_init` will change from 10 to
'auto' in 1.4. Set the value of `n_init` explicitly to suppress the
warning
  warnings.warn(
```



```
# Choose k based on ARI
best_ari_k = 9  # Based on ARI scores
kmeans_ari = KMeans(n_clusters=best_ari_k, random_state=42, n_init=10)
clusters_ari = kmeans_ari.fit_predict(X_train_scaled)

# PCA for visualization
X_train_pca_ari = X_train_pca  # PCA already computed

# Plotting the clusters
plt.figure(figsize=(8, 6))
for cluster in range(best_ari_k):
    indices = clusters_ari == cluster
    plt.scatter(X_train_pca_ari[indices, 0], X_train_pca_ari[indices,
1], label=f'Cluster {cluster}', alpha=0.6)

plt.title('K-Means Clustering Visualization (ARI)')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.legend(title='Cluster')
plt.show()
```



ARI with increase of k

The optimal was 9.

- ARI measures the similarity between clustering results and the ground truth labels. As the number of clusters increase the followng trend is observed.
  - **Plateau**: Beyond the optimal K, the Ari scrore declines as the clusters become too granular, this leads reduction of alignment with the ground truth.
  - **Fluctuations**: As K increases, the ARI fluctuares due to creation of more clustres that do not correspond well to the true data distributions or overfitting.

Prototype Selection using K-means Clustering.

1. Random Selection

```
import numpy as np
import pandas as pd
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
```

```python
def random_prototype_selection(X, y, n_samples):
    """
    Selects a random subset from the data. Train a logistic regression
model
    on the selected data.

    Args:
        X (pd.DataFrame): The input features.
        y (pd.Series): The target labels.
        n_samples(int): The number of samples to select from each
class.

    Returns:
        tuple: A tuple containing the selected features (X_selected)
and labels (y_selected).
    """
    X_selected = []
    y_selected = []

    # Iterate over each unique class
    for label in np.unique(y):
        # Select random samples for the current class
        indices = np.random.choice(np.where(y == label)[0], n_samples,
replace=False)
        X_selected.append(X[indices])
        y_selected.append(y[indices])

    # Concatenate the selected samples
    X_selected = np.vstack(X_selected)
    y_selected = np.concatenate(y_selected)

    return X_selected, y_selected


n_repetitions = 10
accuracies = []
n_samples = 120

# Calculate the accuracy for the randomly selected prototypes over 10
experiments
for _ in range(n_repetitions):
    X_selected, y_selected =
random_prototype_selection(X_train_scaled, y_train[0].values,
n_samples)

    # Train Logistic Regression model
    logistic_regression = LogisticRegression(random_state=42,
max_iter=1000)
    logistic_regression.fit(X_selected, y_selected)
    # Make predictions and calculate accuracy
    y_pred = logistic_regression.predict(X_test_scaled)
    accuracy = accuracy_score(y_test[0].values, y_pred)
    accuracies.append(accuracy)

average_accuracy = np.mean(accuracies)
print(f"Average Accuracy with Random Selection over {n_repetitions}
repetitions: {average_accuracy:.4f}")
```

```
Average Accuracy with Random Selection over 10 repetitions: 0.9192
```

2. K-means Clustering by Class

```
# 2. K-means Clustering by Class
def kmeans_prototype_selection(X, y, n_prototypes_per_class):
    """
    Selects prototypes using K-means clustering for each class.

    Args:
        X (pd.DataFrame): The input features.
        y (pd.Series): The target labels.
        n_prototypes_per_class (int): The number of prototypes to
select from each class.

    Returns:
        pd.DataFrame: The selected prototypes.
        pd.Series: The selected labels.
    """

    #Initialize lists to store selected prototypes and labels
    X_selected = []  # List to store selected feature subsets for each
class
    y_selected = []  # List to store selected labels for each class

    # Step 1: Iterate over each unique class label in the target
labels
    for label in np.unique(y):
        # Filter data points belonging to the current class
        class_data = X.loc[y.values.flatten() == label]

        # Step 2: Cluster the points using K-means with k =
n_prototypes_per_class
        kmeans = KMeans(n_clusters=n_prototypes_per_class,
random_state=42)
        kmeans.fit(class_data)

        # Step 3: Find the closest points to each centroid
        centroids = kmeans.cluster_centers_
        for centroid in centroids:
            distances = np.linalg.norm(class_data.values - centroid,
axis=1)
            closest_point_idx = np.argmin(distances)

X_selected.append(class_data.iloc[closest_point_idx].values)
            y_selected.append(label)

    # Convert lists to numpy arrays
    X_selected = np.array(X_selected)
    y_selected = np.array(y_selected)

    return X_selected, y_selected



# Select prototypes using K-means
y_train = y_train.rename(columns={0: 'Activity'})

X_train_selected_kmeans, y_train_selected_kmeans =
kmeans_prototype_selection(X_train, y_train['Activity'], 20)

# Train Logistic Regression model
logistic_regression_kmeans = LogisticRegression(random_state=42,
max_iter=1000)
logistic_regression_kmeans.fit(X_train_selected_kmeans,
y_train_selected_kmeans)

# Make predictions and calculate accuracy
y_pred_kmeans = logistic_regression_kmeans.predict(X_test)
accuracy_kmeans = accuracy_score(y_test, y_pred_kmeans)
print(f"Accuracy with K-means Selection: {accuracy_kmeans:.4f}")
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/
_kmeans.py:870: FutureWarning: The default value of `n_init` will
change from 10 to 'auto' in 1.4. Set the value of `n_init` explicitly
to suppress the warning
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870
: FutureWarning: The default value of `n_init` will change from 10 to
'auto' in 1.4. Set the value of `n_init` explicitly to suppress the
warning
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870
: FutureWarning: The default value of `n_init` will change from 10 to
'auto' in 1.4. Set the value of `n_init` explicitly to suppress the
warning
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870
```

```
: FutureWarning: The default value of `n_init` will change from 10 to
'auto' in 1.4. Set the value of `n_init` explicitly to suppress the
warning
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870
: FutureWarning: The default value of `n_init` will change from 10 to
'auto' in 1.4. Set the value of `n_init` explicitly to suppress the
warning
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870
: FutureWarning: The default value of `n_init` will change from 10 to
'auto' in 1.4. Set the value of `n_init` explicitly to suppress the
warning
  warnings.warn(

Accuracy with K-means Selection: 0.8931
```

Comparizon of the Random selection and the K-meands clusering by class model Accuracy

Random Selection: Average Accuracy with Random Selection over 10 repetitions: 0.9192.

Using K-means clustering by Class : Accuracy with K-means Selection: 0.8931

The Random prototype selection had a higher Accuracy of 92.23% compared to the K-means clustering that had 90.57%, this could be becaue Random seelction relies on randomly selected prototypes that may capture diverse sample from each class while K-means clustering selects the prototype based on clusters that ensure that the selected samples are representative of the cluser centroids however, this may miss some outlier or diverse samples. And since the later focuess more on centroirds rather than capturing the full variablity of the data this may be the reason of the less performance of the models compared to the reandom selection model.

Autoencoder for Features Learning.

####1. Data Preparation:

```
import glob
import numpy as np

# Load data with proper tensor formatting
def load_inertial_data(path):
    files = glob.glob(path)
    data_dict = {}
    for f in files:
        name = f.split('/')[-1][:-4]
        # Read as numpy array and convert to float32
        data_dict[name] = pd.read_csv(f, sep='\s+',
header=None).values.astype(np.float32)
    return data_dict
```

```
# Load training data
train_data = load_inertial_data("UCI HAR Dataset/train/Inertial
Signals/*.txt")
train_labels = pd.read_csv("UCI HAR Dataset/train/y_train.txt",
header=None)[0].values

# Load Test data
test_data = load_inertial_data("UCI HAR Dataset/test/Inertial
Signals/*.txt")
test_labels = pd.read_csv("UCI HAR Dataset/test/y_test.txt",
header=None)[0].values


print(f"Train Data Dictionary keys: {list(train_data.keys())}")
print(f"For each sensor the Data shape:
{train_data['body_acc_x_train'].shape}")
```

```
Train Data Dictionary keys: ['body_acc_z_train', 'body_gyro_z_train',
'total_acc_z_train', 'body_acc_y_train', 'total_acc_y_train',
'body_acc_x_train', 'total_acc_x_train', 'body_gyro_x_train',
'body_gyro_y_train']
For each sensor the Data shape: (7352, 128)
```

```
from torch.utils.data import Dataset, DataLoader
import torch

# Create PyTorch Dataset

class SensorsDataset(Dataset):
    def __init__(self, data_dict, labels):
        # Stack all signals along the feature dimension  Shape:
(num_samples, 128, num_features)
        self.data = torch.tensor(
            np.stack([data_dict[key] for key in data_dict.keys()],
axis=-1), dtype=torch.float32
        )  # Shape: (num_samples, 128, num_features)
        self.labels = torch.tensor(labels, dtype=torch.long)

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        return self.data[idx], self.labels[idx]

# Create dataset and dataloader
train_dataset = SensorsDataset(train_data, train_labels)
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)

# Verify shapes
sample, label = next(iter(train_loader))
```

```
print(f"Input shape: {sample.shape}")  # Should be (batch_size, 128, 
9)
print(f"Label shape: {label.shape}")   # Should be (batch_size)
```

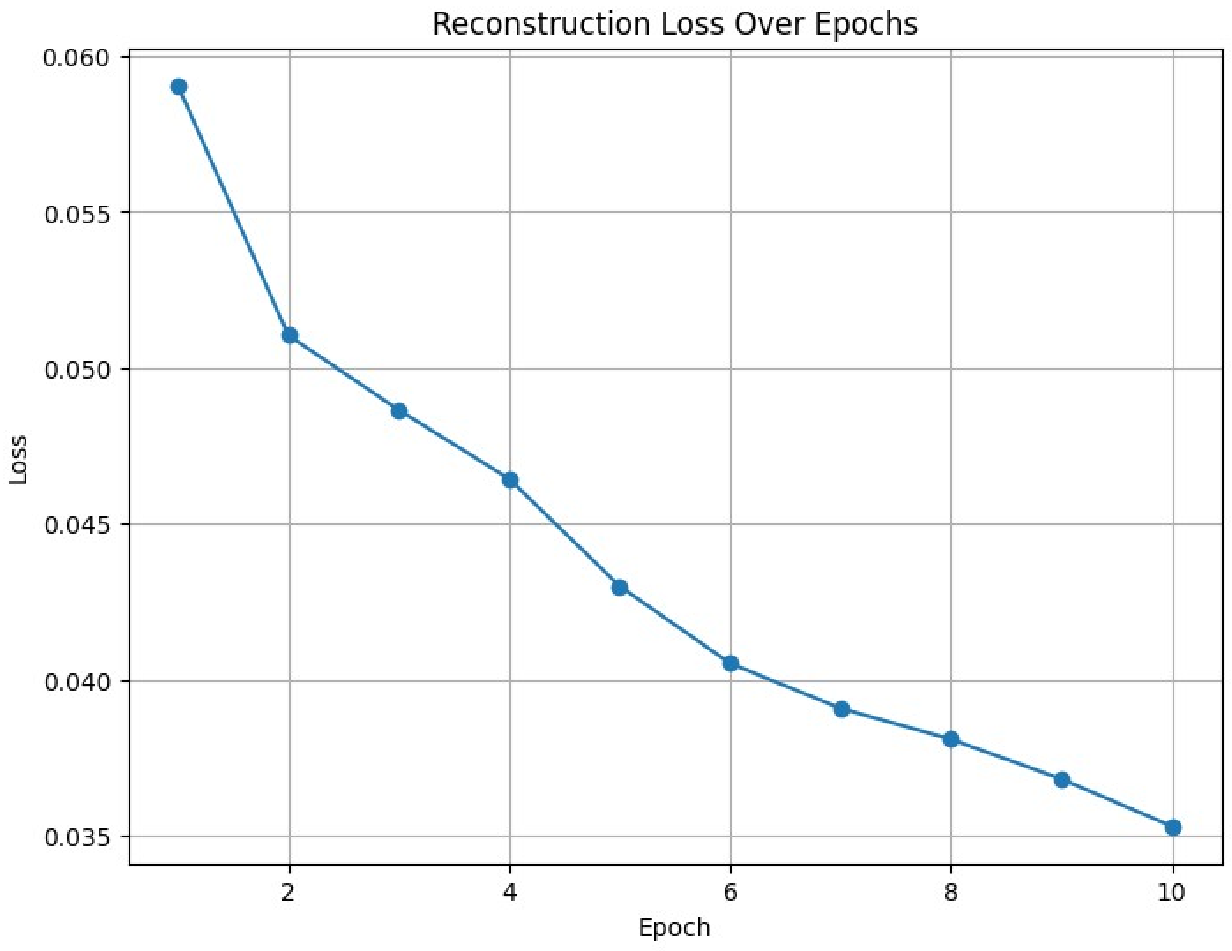
```
Input shape: torch.Size([32, 128, 9])
Label shape: torch.Size([32])
```

2. Autoencoder Implementation

```
import torch.nn as nn
import torch.optim as optim

# 2. Autoencoder Implementation
class TimeSeriesAE(nn.Module):
    def __init__(self, input_size=9, hidden_size=64, encoding_dim=64):
        super().__init__()
        # Encoder
        self.encoder = nn.GRU(input_size, hidden_size, 
batch_first=True, bidirectional=True)
        self.enc_fc = nn.Linear(hidden_size * 2, encoding_dim)

        # Decoder
        self.dec_fc = nn.Linear(encoding_dim, hidden_size * 2)
        self.decoder = nn.GRU(hidden_size * 2, hidden_size, 
batch_first=True, bidirectional=True)
        self.output_layer = torch.nn.Linear(hidden_size * 2, 
input_size)

    def forward(self, x):
        _, hidden = self.encoder(x)
        hidden = torch.cat([hidden[-2], hidden[-1]], dim=1)  # Combine 
bidirectional
        encoded = self.enc_fc(hidden)

        # Decoding
        decoded = self.dec_fc(encoded).unsqueeze(1).repeat(1, 
x.size(1), 1)
        out, _ = self.decoder(decoded)
        reconstructed = self.output_layer(out)

        return reconstructed, encoded

# Instantiate the model
input_size = 9  # Number of features
hidden_size = 64

model = TimeSeriesAE(input_size, hidden_size)
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

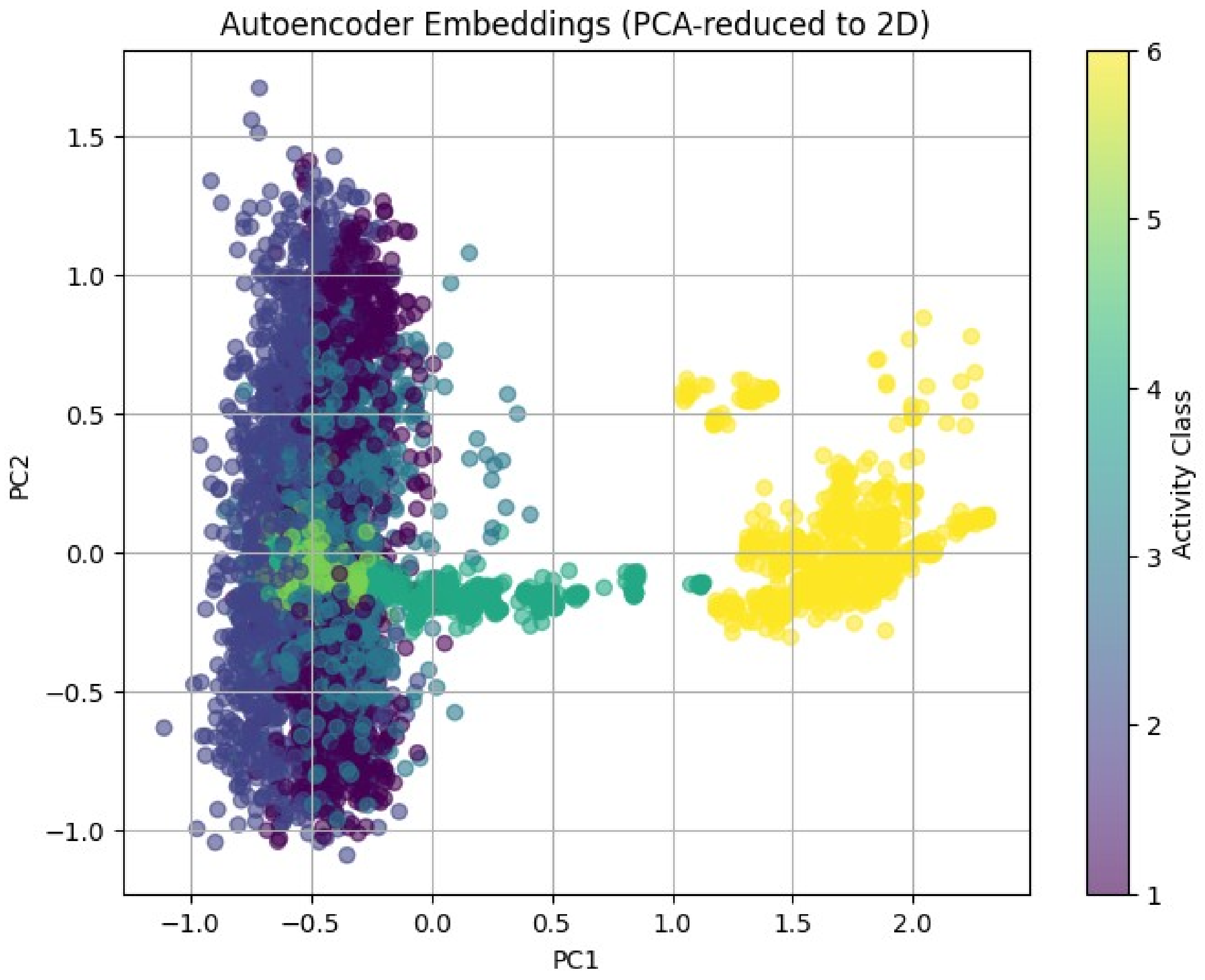
```python
# Train loop for the autoencoder
loss_history = []
num_epochs = 10

for epoch in range(num_epochs):
    model.train()
    total_loss = 0
    for batch_X, _ in train_loader:
        optimizer.zero_grad()
        recon, _ = model(batch_X)
        loss = criterion(recon, batch_X)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()

    avg_loss = total_loss / len(train_loader)
    loss_history.append(avg_loss)
    print(f"Epoch {epoch+1}/{num_epochs}, Loss: {avg_loss:.4f}")

# Plotting the reconstruction loss vs epoch
plt.figure(figsize=(8, 6))
plt.plot(range(1, num_epochs + 1), loss_history, marker='o')
plt.title("Reconstruction Loss Over Epochs")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.grid(True)
plt.show()
```

```
Epoch 1/10, Loss: 0.0591
Epoch 2/10, Loss: 0.0510
Epoch 3/10, Loss: 0.0486
Epoch 4/10, Loss: 0.0464
Epoch 5/10, Loss: 0.0430
Epoch 6/10, Loss: 0.0405
Epoch 7/10, Loss: 0.0391
Epoch 8/10, Loss: 0.0381
Epoch 9/10, Loss: 0.0368
Epoch 10/10, Loss: 0.0353
```



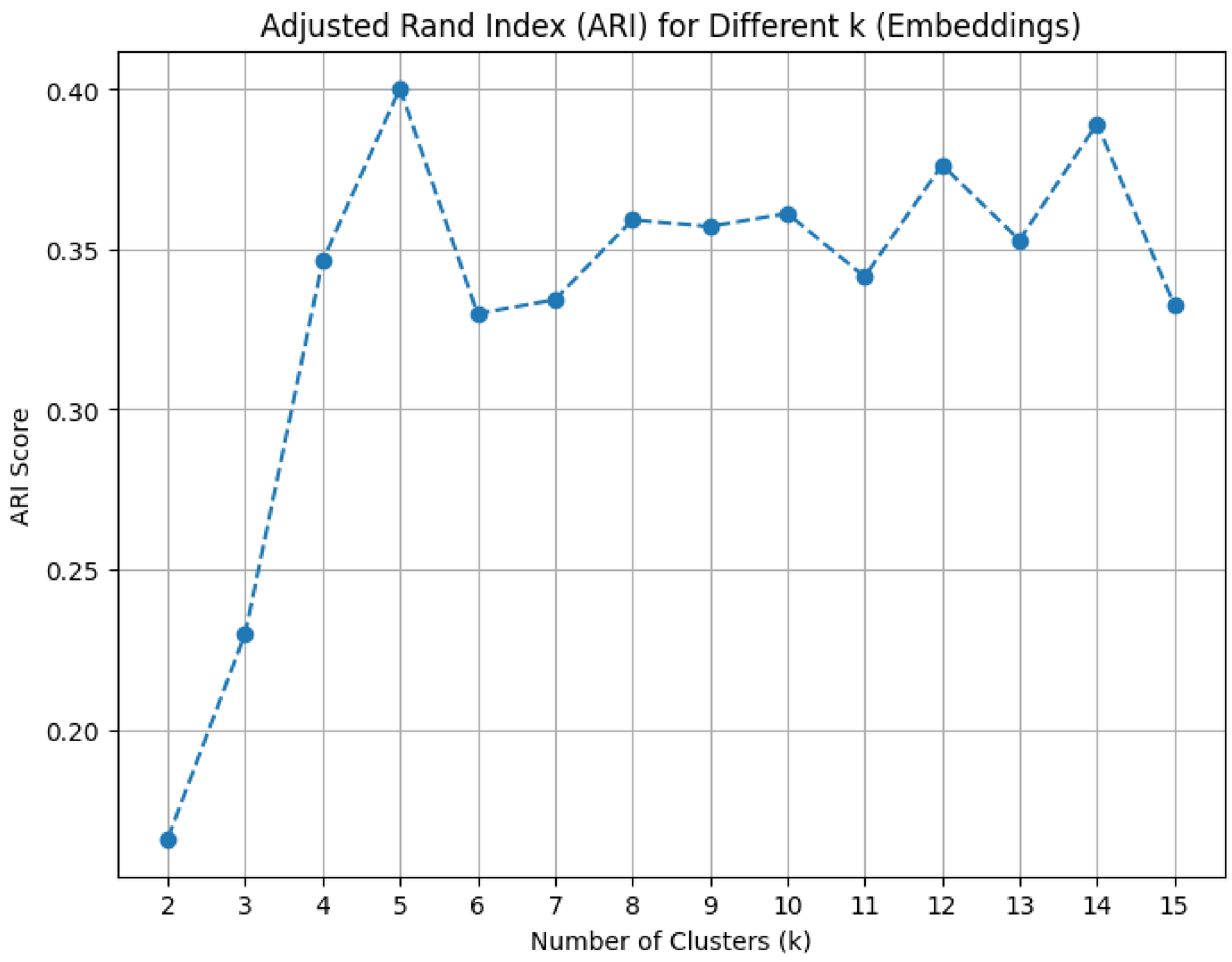
```
# 3. Embedding Extraction and Visualization

ae_loader = DataLoader(train_dataset, batch_size=32, shuffle=False)

# Extract embeddings for the training data
model.eval()
embeddings = []
train_labels = []
with torch.no_grad():
    for batch_X, labels in ae_loader:
        _, encoded = model(batch_X)
        embeddings.append(encoded.numpy())
        train_labels.extend(labels.numpy())

embeddings = np.concatenate(embeddings, axis=0)
train_labels = np.array(train_labels)

# Reduce embeddings to 2D using PCA for visualization
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
embeddings_2d = pca.fit_transform(embeddings)
```

```
# Create a scatter plot of the 2D embeddings
plt.figure(figsize=(8, 6))
activities = np.unique(train_labels)
scatter = plt.scatter(embeddings_2d[:, 0], embeddings_2d[:, 1],
c=train_labels, cmap='viridis', alpha=0.6)
plt.colorbar(scatter, label='Activity Class')
plt.title('Autoencoder Embeddings (PCA-reduced to 2D)')
plt.xlabel("PC1")
plt.ylabel("PC2")
plt.grid(True)
plt.show()
```

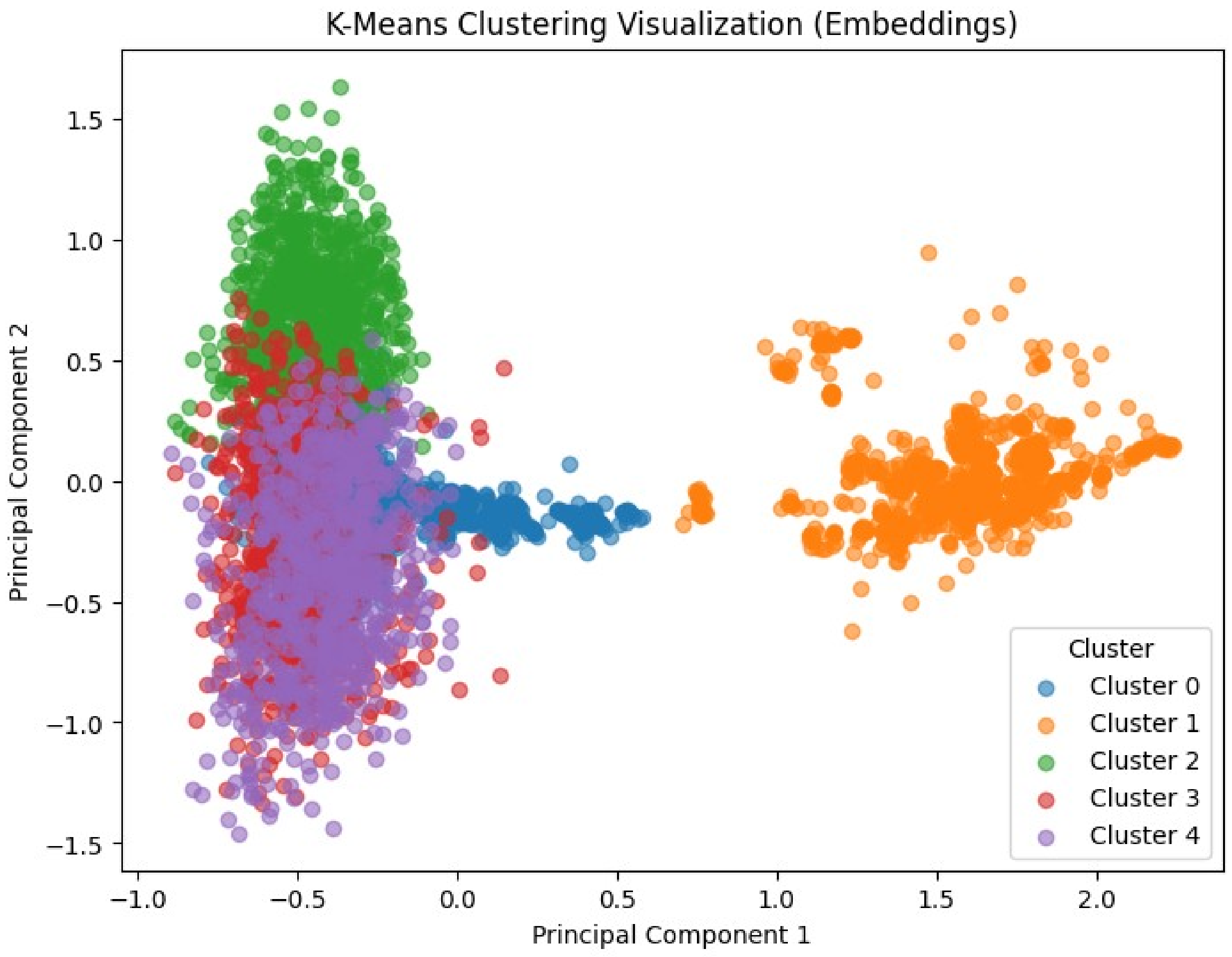


```
from sklearn.metrics import adjusted_rand_score

# Adjusted Rand Index (ARI) for the embeddings


ari_scores = []
```

```
for k in range(2, 16):
    kmeans = KMeans(n_clusters=k, random_state=42, n_init=10)
    clusters = kmeans.fit_predict(embeddings)  # Use embeddings for
clustering
    ari = adjusted_rand_score(train_labels, clusters)  # Compare with
ground truth labels
    ari_scores.append(ari)

# Plotting ARI Scores
plt.figure(figsize=(8, 6))
plt.plot(range(2, 16), ari_scores, marker='o', linestyle='--')
plt.title('Adjusted Rand Index (ARI) for Different k (Embeddings)')
plt.xlabel('Number of Clusters (k)')
plt.ylabel('ARI Score')
plt.xticks(range(2, 16))
plt.grid()
plt.show()
```



```
# Choose k based on ARI
best_embedd_ari_k = 5  # Based on ARI scores or other criteria
kmeans_ari = KMeans(n_clusters=best_embedd_ari_k, random_state=42,
n_init=10)
clusters_ari = kmeans_ari.fit_predict(embeddings)  # Use embeddings
for clustering

# PCA for visualization
pca = PCA(n_components=2)  # Reduce to 2 dimensions for visualization
X_train_pca_ari = pca.fit_transform(embeddings)

# Plotting the clusters
plt.figure(figsize=(8, 6))
for cluster in range(best_embedd_ari_k):
    indices = clusters_ari == cluster
    plt.scatter(X_train_pca_ari[indices, 0], X_train_pca_ari[indices,
1], label=f'Cluster {cluster}', alpha=0.6)

plt.title('K-Means Clustering Visualization (Embeddings)')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.legend(title='Cluster')
plt.show()
```


K-Means Clustering Visualization (Embeddings)
Principal Component 1
Principal Component 2
Cluster
Cluster 0
Cluster 1
Cluster 2
Cluster 3
Cluster 4
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0