Homework 4

Name: Edward Ajayi

Andrew ID: eaajayi

Program: MS EAI

Institution: Carnegie Mellon University Africa

Semester: Spring 2025

Unit: 18-661 - Introduction to Machine Learning for Engineers

Faculty: Prof. Carlee Joe-Wong and Prof. Gauri Joshi

Date: April 19, 2025

Question 1: Distributed SGD

We are given that each worker node i takes a time $X_i \sim \text{Exponential}(\lambda = 2)$ to compute its gradient. The PDF of X is:

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

(a) CDF of $f_X(x)$

The cumulative distribution function (CDF) of X is obtained by integrating the PDF:

$$F_X(x) = \int_0^x \lambda e^{-\lambda t} dt = \left[-e^{-\lambda t}\right]_0^x = 1 - e^{-\lambda x}$$

For $\lambda = 2$:

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

(b) CDF and Expected Value of the Maximum $X_{m:m}$

Let $X_{m:m} = \max(X_1, \ldots, X_m)$ where $X_i \sim \text{Exponential}(\lambda)$ are i.i.d.

CDF: Since the maximum is less than x if and only if all X_i are less than x:

$$F_{X_{m:m}}(x) = \mathbb{P}(X_1 \leq x, \ldots, X_m \leq x) = [F_X(x)]^m = \left(1 - e^{-\lambda x}\right)^m$$

Expected Value: The expected value of the maximum of m i.i.d. exponential variables is:

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

where H_m is the m-th harmonic number and $\lambda = 2$.

$$\boxed{F_{X_{m:m}}(x) = \left(1 - e^{-2x}\right)^m, \quad \mathbb{E}[X_{m:m}] = \frac{1}{2}\sum_{k=1}^m \frac{1}{k}}$$

(c) CDF and Expected Value of the Minimum $X_{1:m}$

Let $X_{1:m} = \min(X_1, \ldots, X_m)$.

CDF: The minimum is less than x if at least one X_i is:

$$F_{X_{1:m}}(x) = 1 - \mathbb{P}(X_1 > x, \ldots, X_m > x) = 1 - (1 - F_X(x))^m = 1 - e^{-m\lambda x}$$

So $X_{1:m} \sim \text{Exponential}(m\lambda)$.

Expected Value:

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

$$\boxed{F_{X_{1:m}}(x) = 1 - e^{-2mx}, \quad \mathbb{E}[X_{1:m}] = \frac{1}{2m}}$$

(d) Simulating Average Runtimes

5000 iterations of training for $m = 1$ to 20 was simulated and the synchronous and asynchronous runtimes were plotted as see in Figure 1 below.

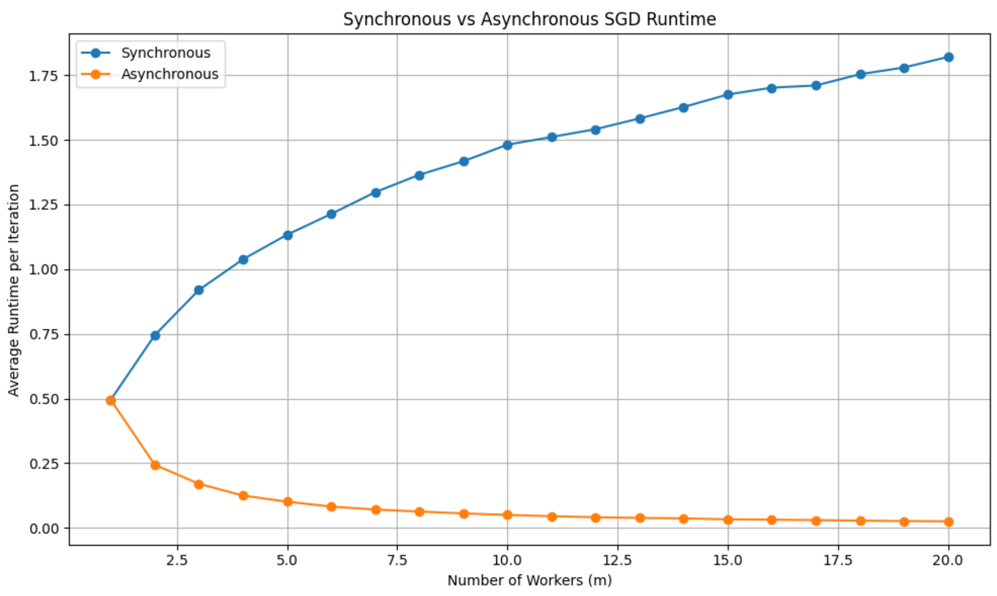


Figure 1: Simulated runtimes of synchronous and asynchronous SGD with varying workers m.

Observation: As m increases: Synchronous SGD runtime increases and asynchronous SGD runtime decreases, as it uses the fastest worker. This trend reflects the expected behavior: synchronous training is bottlenecked by the slowest worker, while asynchronous training benefits from faster updates driven by the quickest worker.

(e) Theoretical Expected Runtimes

From parts (b) and (c):

- **Synchronous SGD:**

$$\mathbb{E}_{\text{sync}}(m) = \mathbb{E}[X_{m:m}] = \frac{1}{\lambda}\sum_{k=1}^{m}\frac{1}{k}$$

For $\lambda = 2$:

$$\boxed{\mathbb{E}_{\text{sync}}(m) = \frac{1}{2}\sum_{k=1}^{m}\frac{1}{k}}$$

- **Asynchronous SGD:**

$$\mathbb{E}_{\text{async}}(m) = \mathbb{E}[X_{1:m}] = \frac{1}{m\lambda}$$

For $\lambda = 2$:

$$\boxed{\mathbb{E}_{\text{async}}(m) = \frac{1}{2m}}$$

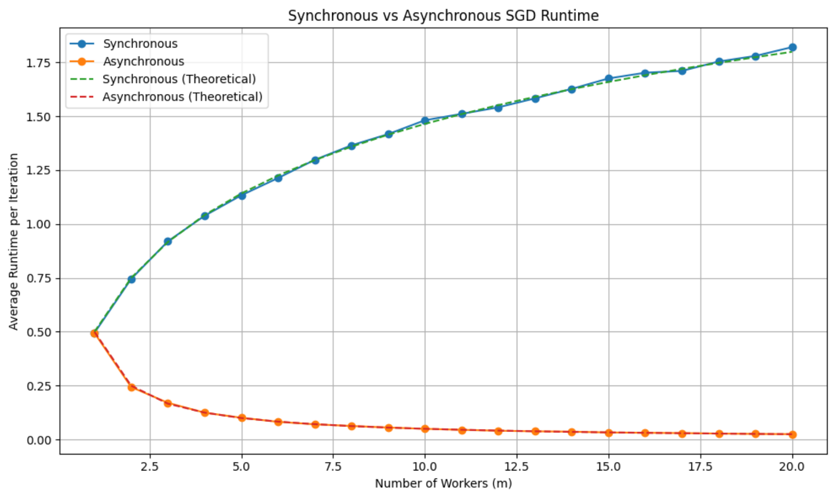


Figure 2: Simulated vs. theoretical runtimes for synchronous and asynchronous SGD.

As shown in Figure 2, the theoretical values closely match the averages from simulation for all $m \in [1, 20]$. The asynchronous curve drops smoothly as $\frac{1}{2m}$, while the synchronous curve rises slowly due to the logarithmic growth of the harmonic sum. This confirms that our simulation aligns with theoretical expectations and validates the correctness of both runtime models.

Question 2: K-means

(a) Proving that μ_k is the mean of the data points assigned to cluster k.

We are given the distortion objective:

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

Assume that the cluster assignments $r_{nk} \in \{0, 1\}$ are known and fixed. For each cluster k, we want to find the μ_k that minimizes D.

We isolate the portion of the objective that involves μ_k:

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

To minimize D_k, we differentiate with respect to μ_k:

$$\frac{\partial D_k}{\partial \mu_k} = \sum_{n=1}^{N} r_{nk} \cdot (-2x_n + 2\mu_k) = 2\left(\sum_{n=1}^{N} r_{nk}\mu_k - \sum_{n=1}^{N} r_{nk}x_n\right)$$

Setting the derivative to zero:

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

This proves that μ_k is the mean of all points assigned to cluster k.

(b) Cluster centers with feature scaling

Suppose each data point $x_n \in \mathbb{R}^d$ is scaled by a diagonal matrix W such that:

$$x'_n = Wx_n$$

We want to minimize the new distortion measure:

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

We again isolate the portion of the distortion involving μ_k:

$$D_k = \sum_{n=1}^{N} r_{nk} \|Wx_n - \mu_k\|^2$$

Differentiating with respect to μ_k:

$$\frac{\partial D_k}{\partial \mu_k} = 2\left(\sum_{n=1}^{N} r_{nk}\mu_k - \sum_{n=1}^{N} r_{nk} W x_n\right)$$

Setting the derivative to zero:

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

Let $\bar{x}_k$ be the mean of the unscaled data in cluster k:

$$\bar{x}_k = \frac{\sum_{n=1}^{N} r_{nk} x_n}{\sum_{n=1}^{N} r_{nk}}$$

Then the expression for the scaled cluster center becomes:

$$\boxed{\mu_k = W\bar{x}_k = W\left(\frac{\sum_{n=1}^{N} r_{nk} x_n}{\sum_{n=1}^{N} r_{nk}}\right)}$$

Question 3

We are given 3 data points:

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

We want to find the first two principal components of the given data.

(a) Covariance Matrix and Eigen Decomposition

Step 1: Compute the mean

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

Step 2: Center the data

$$x'_1 = x_1 - \bar{x} = [-1, -1, -2], \quad x'_2 = x_2 - \bar{x} = [0, 1, 1], \quad x'_3 = x_3 - \bar{x} = [1, 0, 1]$$

Form the matrix:

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

Step 3: Covariance matrix (unnormalized)

$$C_X = X^T X = \begin{bmatrix} (-1)^2 + 0^2 + 1^2 & (-1)(-1) + 0(1) + 1(0) & (-1)(-2) + 0(1) + 1(1) \\ (-1)(-1) + 0(1) + 1(0) & (-1)^2 + 1^2 + 0^2 & (-1)(-2) + 1(1) + 0(1) \\ (-2)(-1) + 1(0) + 1(1) & (-2)(-1) + 1(1) + 1(0) & (-2)^2 + 1^2 + 1^2 \end{bmatrix}$$

$$C_X = X^T X = \begin{bmatrix} 2 & 1 & 3 \\ 1 & 2 & 3 \\ 3 & 3 & 6 \end{bmatrix}$$

Step 4: Eigenvalues and eigenvectors

$$\text{Eigenvalues: } \lambda_1 = 9, \ \lambda_2 = 1, \ \lambda_3 = 0$$

$$\text{Eigenvectors:}\begin{aligned} \lambda = 9 : &\quad u_1 = \left[\frac{1}{2}, \frac{1}{2}, 1\right]^T \\ \lambda = 1 : &\quad u_2 = [-1, 1, 0]^T \\ \lambda = 0 : &\quad u_3 = [-1, -1, 1]^T \end{aligned}$$

(b) First Two Principal Components

We choose eigenvectors corresponding to the largest two eigenvalues:

$$u_1 = \left[\frac{1}{2}, \frac{1}{2}, 1\right]^T, \quad u_2 = [-1, 1, 0]^T$$

These form the principal component basis for the best-fitting 2D subspace.

(c) Projection onto 2D Subspace

We approximate each data point using:

$$x_i \approx \tilde{x}_i = a_{i1}u_1 + a_{i2}u_2 + \bar{x}$$

Where:

$$a_{i1} = u_1^T(x_i - \bar{x}), \quad a_{i2} = u_2^T(x_i - \bar{x})$$

For x_1':

$$a_{11} = u_1^T x_1' = \left[\frac{1}{2}, \frac{1}{2}, 1\right] \cdot [-1, -1, -2] = -0.5 - 0.5 - 2 = -3$$

$$a_{12} = u_2^T x_1' = [-1, 1, 0] \cdot [-1, -1, -2] = 1 - 1 + 0 = 0$$

$$\Rightarrow \tilde{x}_1 = -3u_1 + \bar{x} = -3 \cdot \left[\frac{1}{2}, \frac{1}{2}, 1\right] + [1, 0, 0] = [-0.5, -1.5, -3]$$

$$\|x_1 - \tilde{x}_1\| = \sqrt{(0 + 0.5)^2 + (-1 + 1.5)^2 + (-2 + 3)^2} = \sqrt{0.25 + 0.25 + 1} = \sqrt{1.5}$$

For $x_2' = [0, 1, 1]$:

$$a_{21} = u_1^T x_2' = 1.5, \quad a_{22} = u_2^T x_2' = 1$$

$$\tilde{x}_2 = 1.5 \cdot \left[\frac{1}{2}, \frac{1}{2}, 1\right] + 1 \cdot [-1, 1, 0] + [1, 0, 0] = [0.75, 1.75, 1.5]$$

$$\|x_2 - \tilde{x}_2\| = \sqrt{(0.25)^2 + (-0.75)^2 + (-0.5)^2} = \sqrt{0.875} \approx 0.935$$

For $x'_3 = [1, 0, 1]$:

$$a_{31} = u_1^T x'_3 = 1.5, \quad a_{32} = u_2^T x'_3 = -1$$

$$\tilde{x}_3 = 1.5 \cdot \left[\frac{1}{2}, \frac{1}{2}, 1\right] - 1 \cdot [-1, 1, 0] + [1, 0, 0] = [2.75, -0.25, 1.5]$$

$$\|x_3 - \tilde{x}_3\| = \sqrt{(-0.75)^2 + (0.25)^2 + (-0.5)^2} = \sqrt{0.875} \approx 0.935$$

Question 4

Q. 4.3

The plot showing the first two principal components can be seen in Figure below

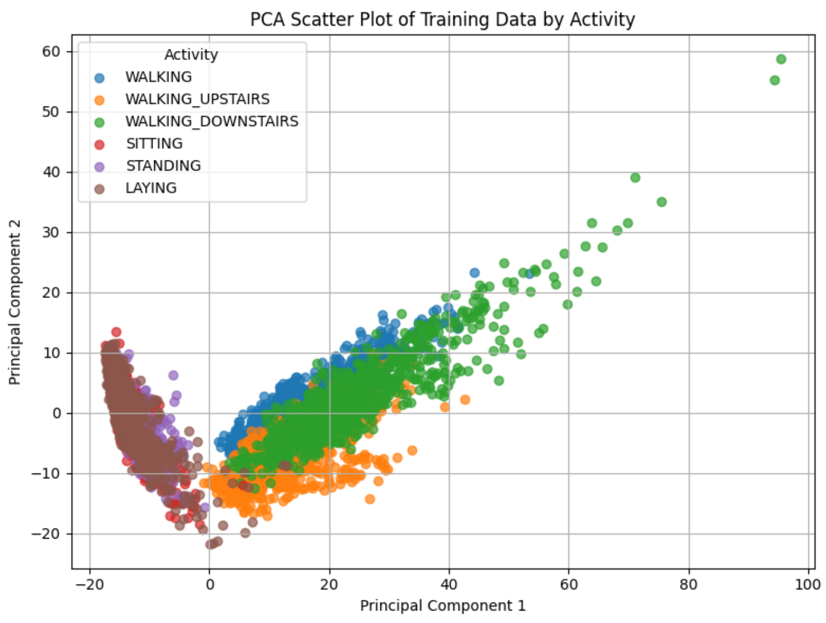


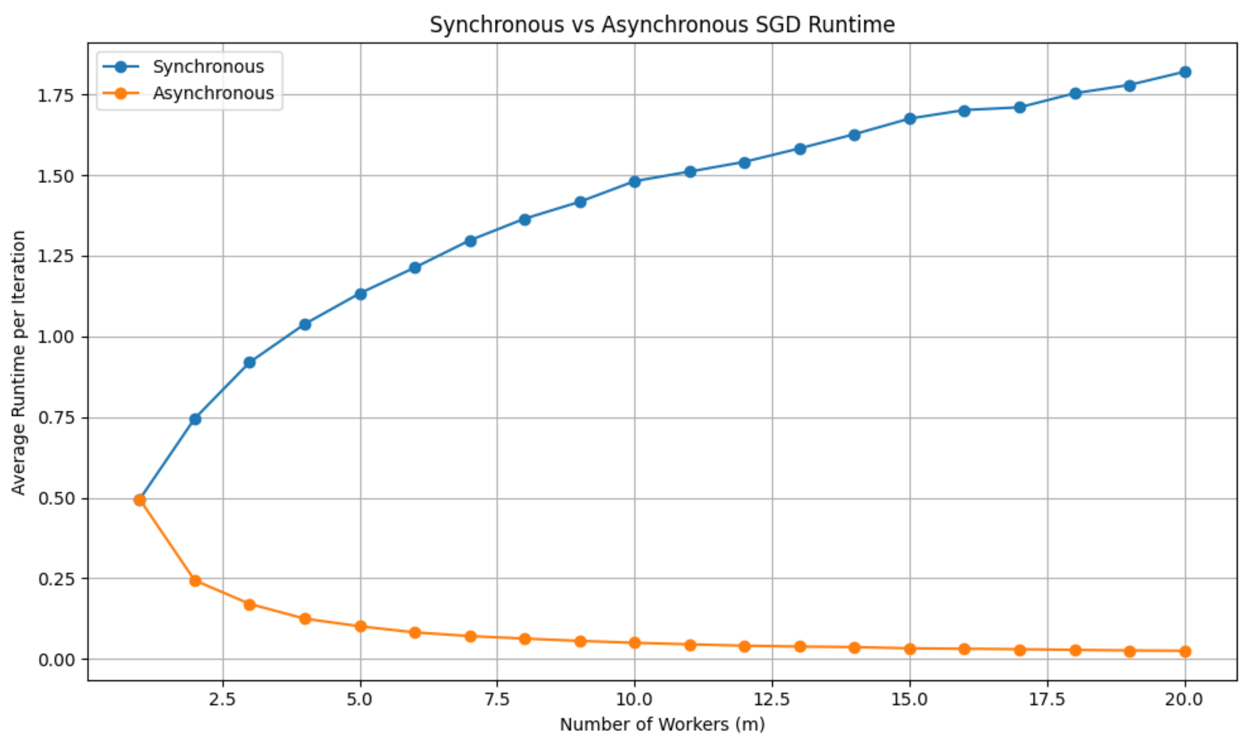
Figure 3: Scatter plot of first two principal components

Q. 4.4d

Clustering with hand-crafted features (Part 4.2) gave a better ARI score (0.442 at k=10) than the autoencoder embeddings (0.4004 at k=5). This shows that the engineered features captured more useful patterns for clustering. The autoencoder might not have learned enough structure from the data or compressed too much. While autoencoders are powerful, in this case, simple statistical features worked better. With more tuning, the autoencoder could still improve. Hand-crafted features explicitly encode domain-relevant statistics, while autoencoder may compress structure not aligned with class separation, leading to lower ARI despite representation learning.

1.4 Synchronous SGD and asynchronous SGD Simulation

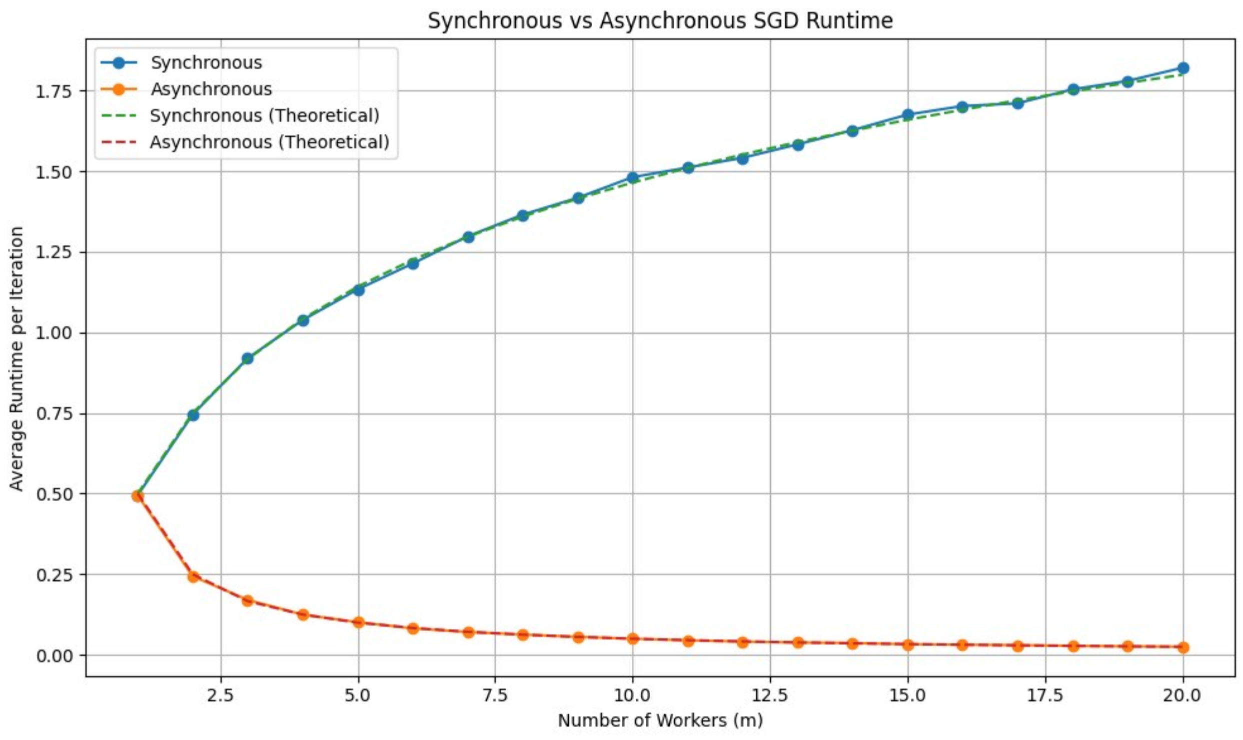
```
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(42)

lambda_param = 2
num_iterations = 5000
m_values = np.arange(1, 21)

sync_runtimes = []
async_runtimes = []

for m in m_values:
    runtimes = np.random.exponential(scale=1/lambda_param, 
size=(num_iterations, m))

    sync_runtime = np.mean(np.max(runtimes, axis=1))
    sync_runtimes.append(sync_runtime)

    async_runtime = np.mean(np.min(runtimes, axis=1))
    async_runtimes.append(async_runtime)

plt.figure(figsize=(10, 6))
plt.plot(m_values, sync_runtimes, marker='o', label='Synchronous')
plt.plot(m_values, async_runtimes, marker='o', label='Asynchronous')
plt.xlabel('Number of Workers (m)')
plt.ylabel('Average Runtime per Iteration')
plt.title('Synchronous vs Asynchronous SGD Runtime')
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()
```



Adding theoretical values

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

np.random.seed(42)

lambda_param = 2
num_iterations = 5000
m_values = np.arange(1, 21)

sync_runtimes = []
async_runtimes = []

for m in m_values:
    runtimes = np.random.exponential(scale=1/lambda_param,
size=(num_iterations, m))

    sync_runtime = np.mean(np.max(runtimes, axis=1))
    sync_runtimes.append(sync_runtime)

    async_runtime = np.mean(np.min(runtimes, axis=1))
    async_runtimes.append(async_runtime)

harmonic_numbers = np.cumsum(1 / m_values)
sync_theory = (1 / lambda_param) * harmonic_numbers
async_theory = 1 / (lambda_param * m_values)
```

```
plt.figure(figsize=(10, 6))
plt.plot(m_values, sync_runtimes, marker='o', label='Synchronous')
plt.plot(m_values, async_runtimes, marker='o', label='Asynchronous')
plt.plot(m_values, sync_theory, '--', label='Synchronous
(Theoretical)')
plt.plot(m_values, async_theory, '--', label='Asynchronous
(Theoretical)')
plt.xlabel('Number of Workers (m)')
plt.ylabel('Average Runtime per Iteration')
plt.title('Synchronous vs Asynchronous SGD Runtime')
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()
```



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.")
```

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

Load the data into a dataframe

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

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

# Load feature names
feature_names = pd.read_csv(features_path, delim_whitespace=True,
header=None)[1].tolist()

# Load training data
X_train = pd.read_csv(os.path.join(train_path, "X_train.txt"),
delim_whitespace=True, header=None)
X_train.columns = feature_names
y_train = pd.read_csv(os.path.join(train_path, "y_train.txt"),
delim_whitespace=True, header=None)
y_train.columns = ['Activity']

# Load testing data
X_test = pd.read_csv(os.path.join(test_path, "X_test.txt"),
delim_whitespace=True, header=None)
X_test.columns = feature_names
y_test = pd.read_csv(os.path.join(test_path, "y_test.txt"),
delim_whitespace=True, header=None)
y_test.columns = ['Activity']

# 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-98-809d1cf9ed68>:13: FutureWarning: The
'delim_whitespace' keyword in pd.read_csv is deprecated and will be
removed in a future version. Use ``sep='\s+'`` instead
  feature_names = pd.read_csv(features_path, delim_whitespace=True,
header=None)[1].tolist()
<ipython-input-98-809d1cf9ed68>:16: 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(os.path.join(train_path, "X_train.txt"),
delim_whitespace=True, header=None)
```

```
<ipython-input-98-809d1cf9ed68>:18: 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(os.path.join(train_path, "y_train.txt"),
delim_whitespace=True, header=None)
<ipython-input-98-809d1cf9ed68>:22: 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 = pd.read_csv(os.path.join(test_path, "X_test.txt"),
delim_whitespace=True, header=None)

First 5 rows of training feature dataframe:

<ipython-input-98-809d1cf9ed68>:24: FutureWarning: The
'delim_whitespace' keyword in pd.read_csv is deprecated and will be
removed in a future version. Use ``sep='\s+'`` instead
  y_test = pd.read_csv(os.path.join(test_path, "y_test.txt"),
delim_whitespace=True, header=None)

{"type":"dataframe","variable_name":"X_train"}
```

scaling the data and PCA

```python
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
# TODO: Scale X_train
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, columns=feature_names)
X_test = pd.DataFrame(X_test_scaled, columns=feature_names)


# 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 = X_train.copy()
training_df['Activity'] = y_train.values

# TODO: Combine X_test and y_test into a single DataFrame named
testing_df.
testing_df = X_test.copy()
testing_df['Activity'] = y_test.values

# 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:
```

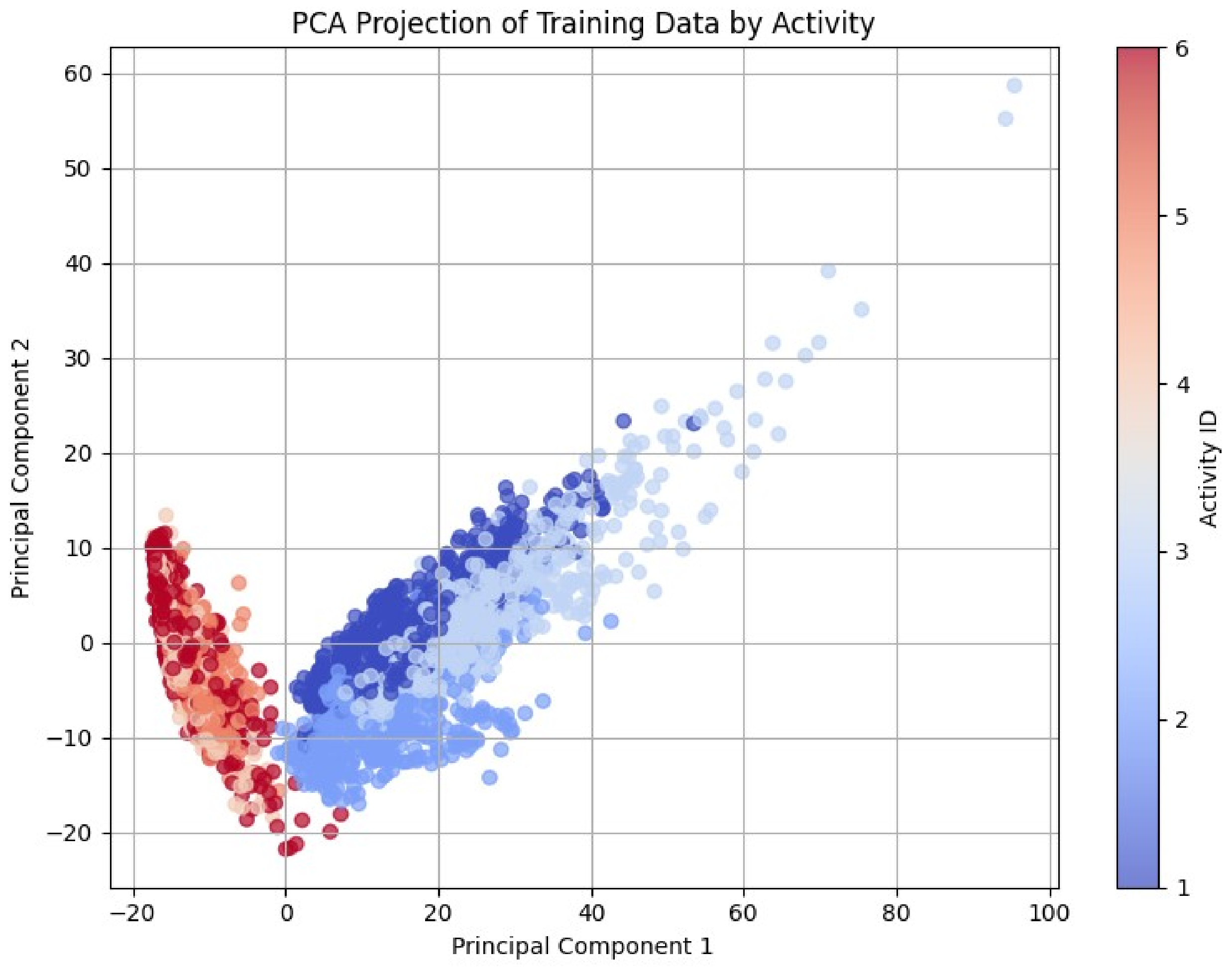
{"type":"dataframe","variable_name":"training_df"}

```python
from sklearn.decomposition import PCA

X_train_only = training_df.drop('Activity', axis=1)
# TODO perform PCA on the train data and get the first 2 PC
pca = PCA(n_components=2)
X_train_pca = pca.fit_transform(X_train_only)
```

Visualize the data

```python
# Visualize training data using PCA


# Use the featre decoder to create Acitivtiy 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']))

 # TODO use the mapping to decode the Activities labels
Activity_Name = training_df['Activity'].replace(activity_mapping)  #
TODO

# TODO: Create a scatter plot using the X_train_pca and the Activity
Names
plt.figure(figsize=(8, 6))
scatter = plt.scatter(
    X_train_pca[:, 0], X_train_pca[:, 1],
    c=training_df['Activity'],
    cmap='coolwarm', alpha=0.7
)

# Plot enhancements
plt.title("PCA Projection of Training Data by Activity")
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.colorbar(scatter, ticks=range(1, 7), label='Activity ID')
plt.grid(True)
plt.tight_layout()
# TODO <--code below-->
```

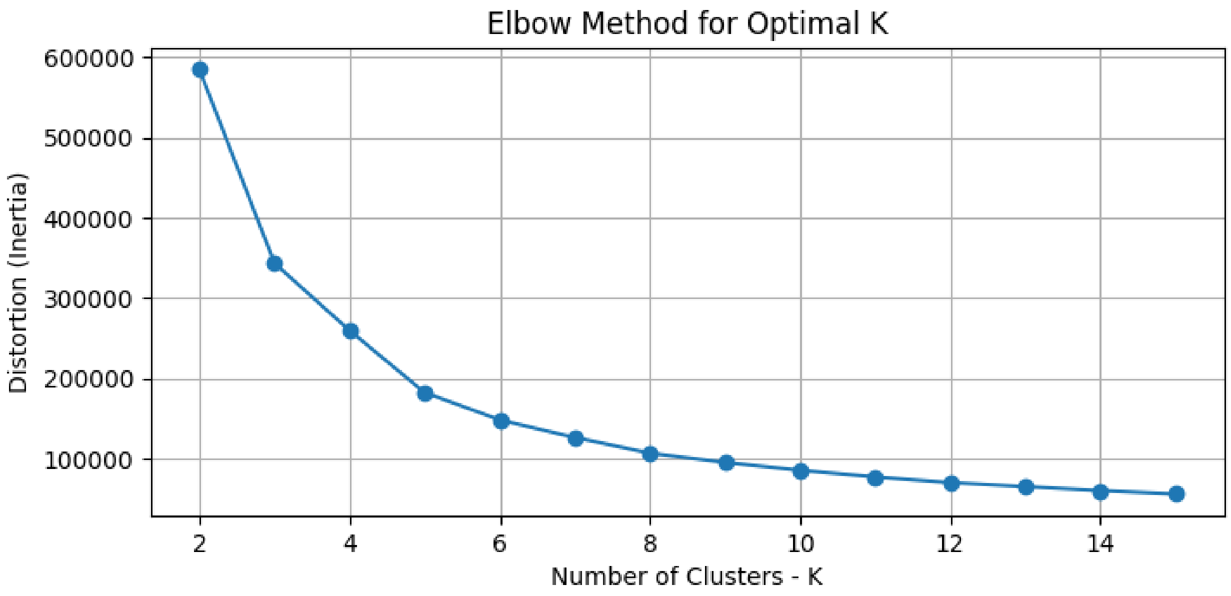
```
plt.show()
```



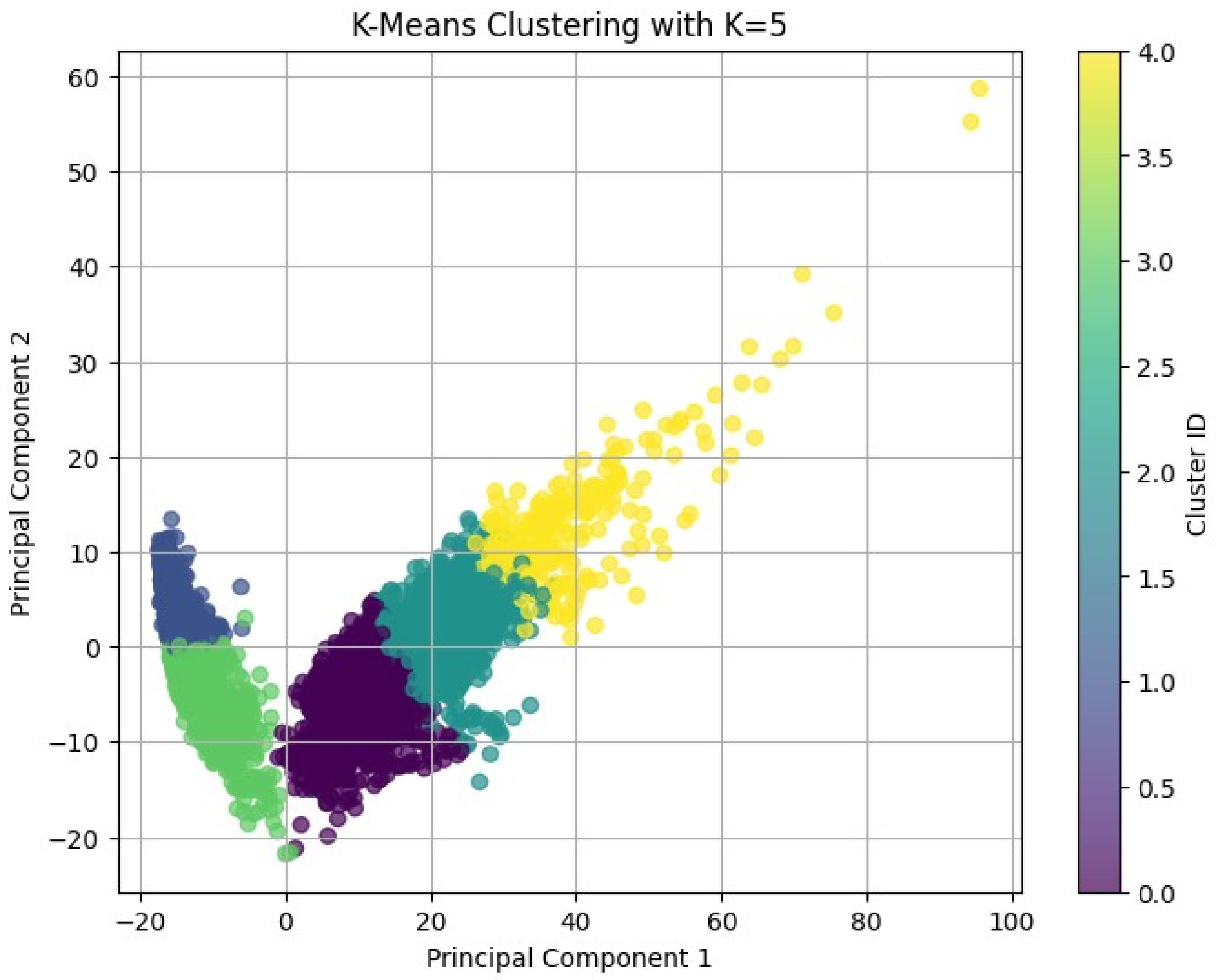
Kmeans Clustering and The Optimal Number of Clusters

1. **Elbow Method**

```
from sklearn.cluster import KMeans
# Elbow Method
distortion_values = []
for k in range(2, 16):
    kmeans = KMeans(n_clusters=k, n_init=10, random_state=42)
    kmeans.fit(X_train_pca)
    distortion_values.append(kmeans.inertia_)  # <-- distortion =
inertia

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



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

# TODO: PCA for visualization
pca = PCA(n_components=2) # TODO
X_train_pca_elbow =  pca.fit_transform(X_train_only) # TODO

# Plotting the clusters
plt.figure(figsize=(8, 6))
# TODO <--code below-->
scatter_elbow = plt.scatter(
    X_train_pca_elbow[:, 0], X_train_pca_elbow[:, 1],
    c=clusters_elbow, cmap='viridis', alpha=0.7
)
plt.title(f"K-Means Clustering with K={elbow_k}")
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.colorbar(scatter_elbow, label='Cluster ID')
plt.grid(True)
plt.show()
```



4.2a - Observation

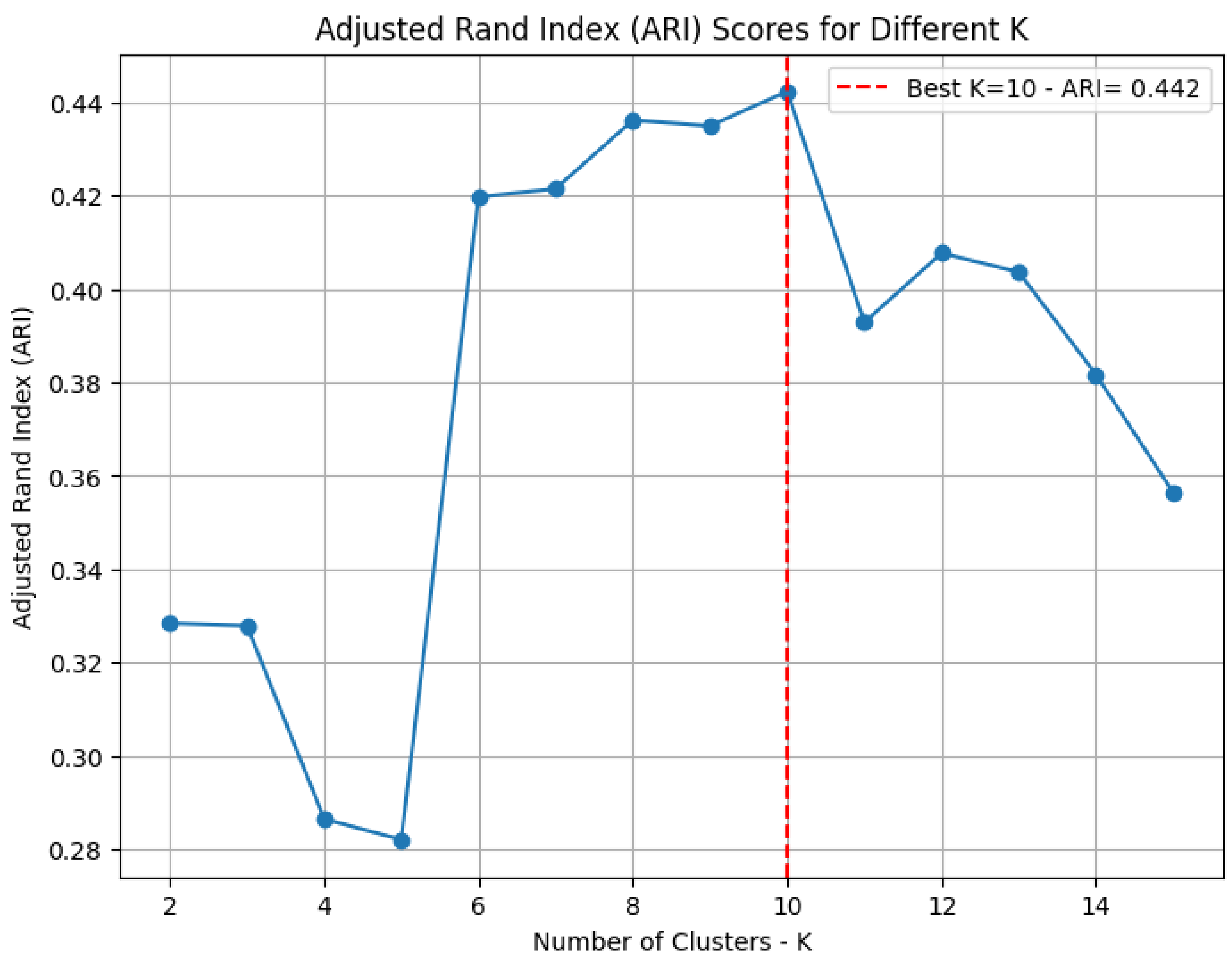
As k increases from 2 to 15, the distortion decreases monotonically. This is expected, as more clusters reduce the average distance between data points and their assigned cluster centers. However, the rate of decrease is steep initially and becomes more gradual after a certain point. Specifically, at k=5, an elbow is noticed on the plot and the distortion still decreases beyond this point, but the marginal improvement diminishes.

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):
    # TODO <--code below-->
    kmeans = KMeans(n_clusters=k, n_init=10, random_state=42)
    # kmeans = KMeans(n_clusters=k,n random_state=0)
    clusters = kmeans.fit_predict(X_train_only)
    ari = adjusted_rand_score(training_df['Activity'].values,
clusters)
```

```
    ari_scores.append(ari)

# Select the best K based on ARI scores
best_k = ari_scores.index(max(ari_scores)) + 2  # +2 because k starts
from 2

print(f"Best K based on ARI scores: {best_k}")

# Plotting ARI Scores
plt.figure(figsize=(8, 6))
# TODO <--code below-->
plt.plot(range(2, 16), ari_scores, marker='o', linestyle='-')
plt.axvline(x=best_k, color='red', linestyle='--', label=f'Best
K={best_k} - ARI= {np.max(ari_scores):.3f}')
plt.title("Adjusted Rand Index (ARI) Scores for Different K")
plt.xlabel("Number of Clusters - K")
plt.ylabel("Adjusted Rand Index (ARI)")
plt.legend()
plt.grid()
plt.show()
```

```
Best K based on ARI scores: 10
```

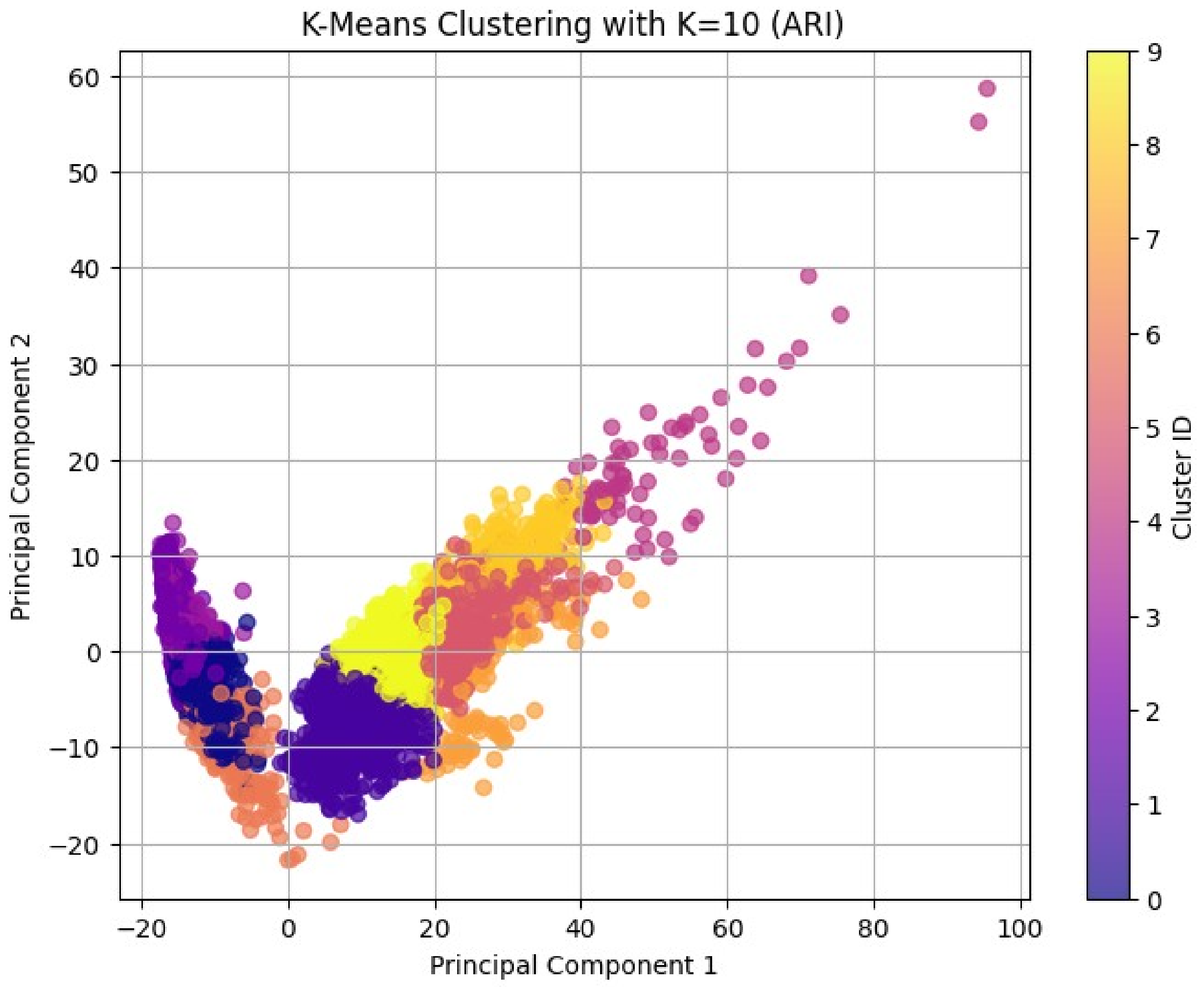


4.2b - Observation

From the plot it can be noticed that k decreased between 2 to 5 then started increasing while peaking at k = 10, suggesting good agreement with true labels then the ARI begins to decline again suggesting that further increasing the number of clusters leads to over-partitioning the data which will lead to reduction in alignment with the ground truth and overfitting.

```
import numpy as np
# Choose k based on ARI
best_ari_k = np.argmax(ari_scores) + 2 # TODO
kmeans_ari = KMeans(n_clusters=best_ari_k, random_state=42, n_init=10)
clusters_ari = kmeans_ari.fit_predict(X_train)

# PCA for visualization
pca =  PCA(n_components=2) # TODO
X_train_pca_ari =  pca.fit_transform(X_train) # TODO


# Plotting the clusters
plt.figure(figsize=(8, 6))
```

```
# TODO <--code below-->
scatter_ari = plt.scatter(
    X_train_pca_ari[:, 0], X_train_pca_ari[:, 1],
    c=clusters_ari, cmap='plasma', alpha=0.7
)
plt.title(f"K-Means Clustering with K={best_ari_k} (ARI)")
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.colorbar(scatter_ari, label='Cluster ID')
plt.grid(True)
plt.show()
```



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

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).
    """
    data = pd.concat([X, y], axis=1)

    selected_samples = pd.DataFrame()

    for class_label in y.unique():
        class_samples = data[data[y.name] == class_label]
        selected = class_samples.sample(n=min(n_samples,
len(class_samples)),
                                        replace=False,

random_state=np.random.RandomState())
        selected_samples = pd.concat([selected_samples, selected])

    # Split back into X and y
    X_selected = selected_samples.drop(columns=[y.name])
    y_selected = selected_samples[y.name]

    return X_selected, y_selected

n_repetitions = 10
accuracies = []
n_samples = 120

# Define X and y based on training_df
X = training_df.drop(columns=['Activity'])  # Features
```

```
y = training_df['Activity']  # Labels

for _ in range(n_repetitions):
    # Select random prototypes
    X_selected, y_selected = random_prototype_selection(X, y,
n_samples)

    # Train logistic regression model
    model = LogisticRegression(max_iter=1000)
    model.fit(X_selected, y_selected)

    # Predict on test set (assuming X_test and y_test are defined)
    y_pred = model.predict(X_test)

    # Calculate accuracy
    acc = accuracy_score(y_test, y_pred)
    accuracies.append(acc)

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: 0.9254
```

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


    # TODO:
      # Step 1: Iterate over each unique class label in the target
```

```
labels
      # Step 2: for each class cluster its points using k =
n_prototypes_per_class
      # Step 3: Find the closest points to each centroid
    # TODO <--code below-->
    for label in np.unique(y):
        # Step 1: Get all samples of this class
        X_class = X[y == label]

        # Step 2: KMeans clustering for this class
        kmeans = KMeans(n_clusters=n_prototypes_per_class,
random_state=42, n_init=10)
        kmeans.fit(X_class)

        # Step 3: Find the closest point to each centroid
        from sklearn.metrics import pairwise_distances_argmin_min
        closest_indices, _ =
pairwise_distances_argmin_min(kmeans.cluster_centers_, X_class)

        # Append selected features and labels
        X_selected.append(X_class.iloc[closest_indices])
        y_selected.append(pd.Series([label] * n_prototypes_per_class))


    return pd.concat(X_selected, ignore_index=True),
pd.concat(y_selected, ignore_index=True)



# Select prototypes using K-means
# X_train_selected_kmeans, y_train_selected_kmeans =
kmeans_prototype_selection(X_train, y_train['Activity'], 20)
X_train_selected_kmeans, y_train_selected_kmeans =
kmeans_prototype_selection(training_df.drop("Activity", axis=1),

training_df["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}")
```

```
Accuracy with K-means Selection: 0.9006
```

Q. 4.3b - Random selection vs K-Means

The model trained on randomly selected prototypes slightly outperformed the one using K-means-based selection, with an average accuracy of 0.9254 compared to 0.9006. While K-means ensures diverse coverage by selecting from distinct clusters, it may miss borderline or high-variance examples that are important for classification. Random selection, especially when repeated, is more likely to include such informative points, which could explain its better performance in this case.

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(train_data.keys())
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}")
```

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

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

# 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
sorted(data_dict.keys())], axis=-1)) # TODO
        self.labels = torch.tensor(labels - 1)  #TODO

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

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

# Create dataset and dataloader
# Ensure that the cell defining `train_data` is executed before
running this cell.
train_dataset = SensorsDataset(train_data, train_labels)
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
# TODO: create pytorch dataloader with Batch sie 32, and shuffle

# 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)
```

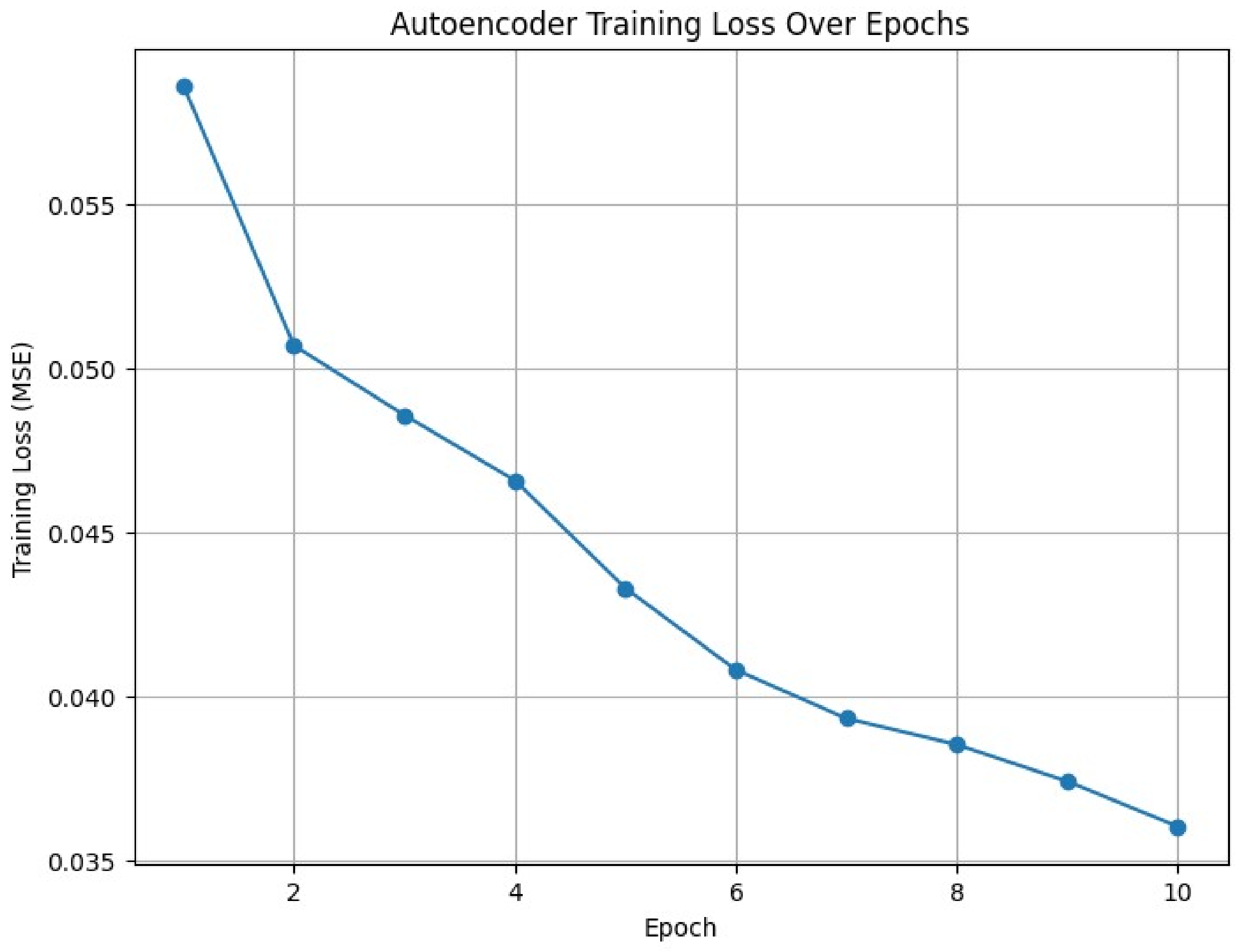
```
Input shape: torch.Size([32, 128, 9])
Label shape: torch.Size([32])
```

2. Autoencoder Implementation

```python
import torch.nn as nn
# 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=input_size,
hidden_size=hidden_size, batch_first=True, bidirectional=True) # TODO:
bidirectional GRU with proper hidden layer size
```

```
        self.enc_fc = nn.Linear(hidden_size * 2, encoding_dim) # TODO:
fully connected layer for the encoder (output encoder_dim)

        # Decoder
        self.dec_fc = nn.Linear(encoding_dim, hidden_size * 2) # TODO:
fully connected layer for the decoder
        self.decoder = nn.GRU(input_size=hidden_size * 2,
hidden_size=hidden_size, batch_first=True, bidirectional=True) # TODO:
bidirectional GRU with proper input and hidden layer size
        self.output_layer = torch.nn.Linear(hidden_size * 2,
input_size) # fully connected layer for the output ( ouput is the
input size)
        # note The input is is hidden_size*2 for bidirectional

    def forward(self, x):
        # Encoding
        _, 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

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

model = TimeSeriesAE(input_size).to(device)

# Define loss function and optimizer
criterion = nn.MSELoss() # TODO
optimizer = torch.optim.Adam(model.parameters(), lr=0.001) # TODO

# TODO: 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:
        # TODO <--code below-->
        batch_X = batch_X.to(device)
        optimizer.zero_grad()
        reconstructed, _ = model(batch_X)
        loss = criterion(reconstructed, 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 accuracy vs epcoh
plt.figure(figsize=(8, 6))
# TODO <--code below-->
# Plotting the loss vs epoch
plt.figure(figsize=(8, 6))
plt.plot(range(1, num_epochs + 1), loss_history, marker='o')
plt.xlabel("Epoch")
plt.ylabel("Training Loss (MSE)")
plt.title("Autoencoder Training Loss Over Epochs")
plt.grid(True)
plt.show()
```

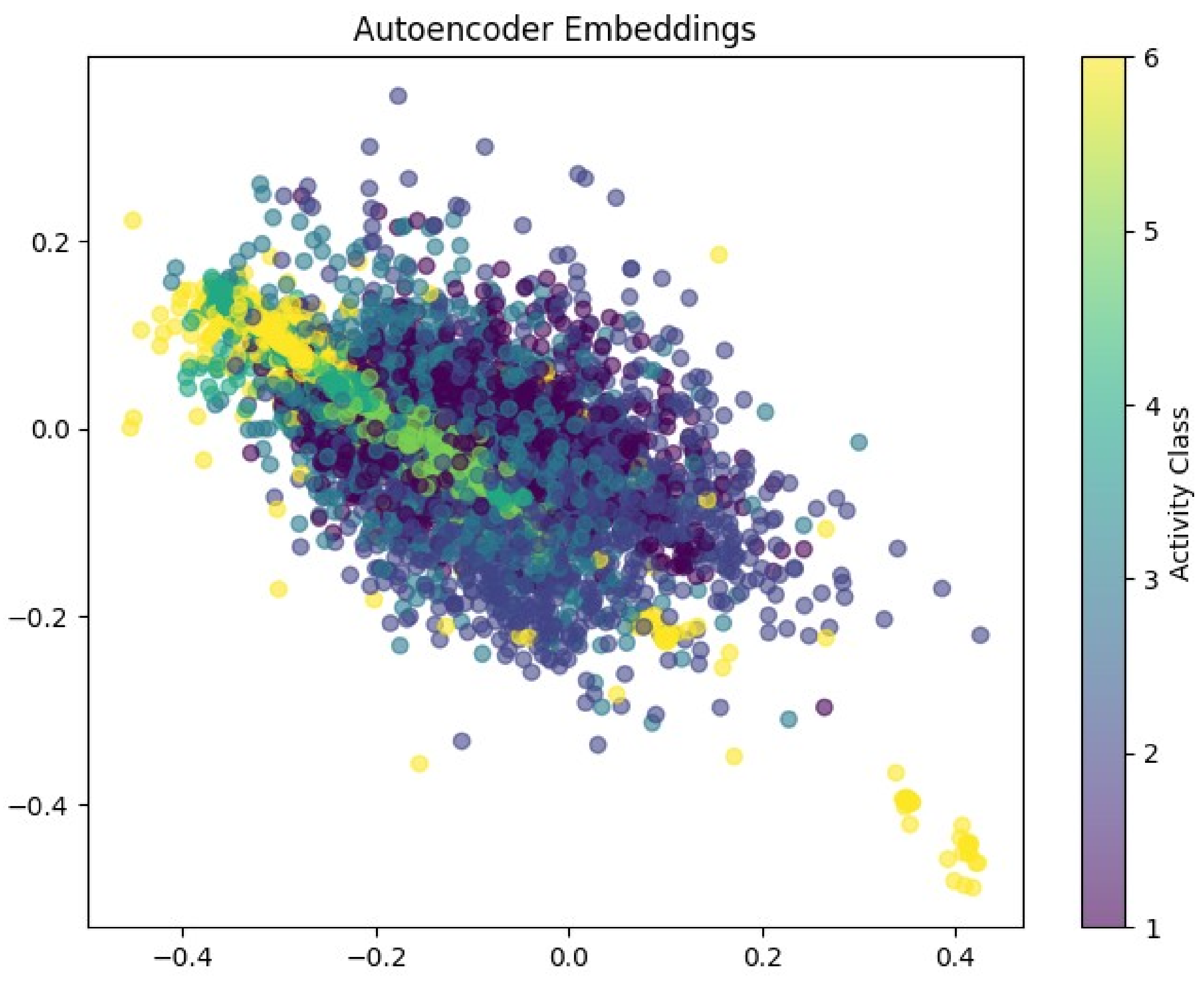
```
Epoch 1/10, Loss: 0.0586
Epoch 2/10, Loss: 0.0507
Epoch 3/10, Loss: 0.0486
Epoch 4/10, Loss: 0.0466
Epoch 5/10, Loss: 0.0433
Epoch 6/10, Loss: 0.0408
Epoch 7/10, Loss: 0.0393
Epoch 8/10, Loss: 0.0385
Epoch 9/10, Loss: 0.0374
Epoch 10/10, Loss: 0.0360

<Figure size 800x600 with 0 Axes>
```



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:
      # TODO <--code below-->
      batch_X = batch_X.to(device)
      _, encoded = model(batch_X)
      embeddings.append(encoded.cpu().numpy())
      train_labels.extend(labels.cpu().numpy())

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

# Create a scatter plot of the 2D embeddings
plt.figure(figsize=(8, 6))
activities = np.unique(y_train)
```

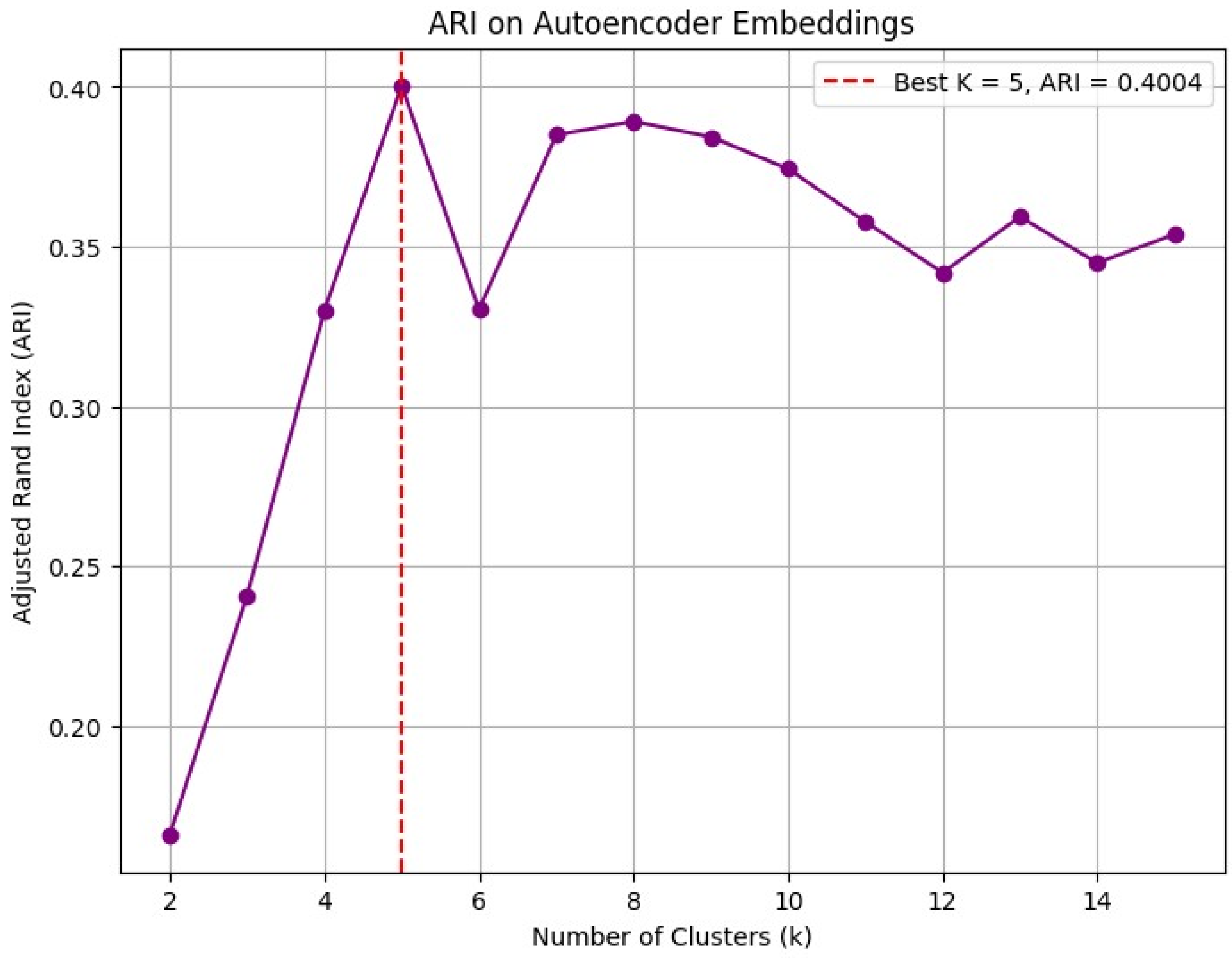
```
plt.scatter(embeddings[:, 0], embeddings[:, 1], c=y_train.values,
cmap='viridis', alpha=0.6)
plt.colorbar(label='Activity Class')
plt.title('Autoencoder Embeddings')
plt.show()
```



4. Adjusted Rand Index (ARI) for the embeddings

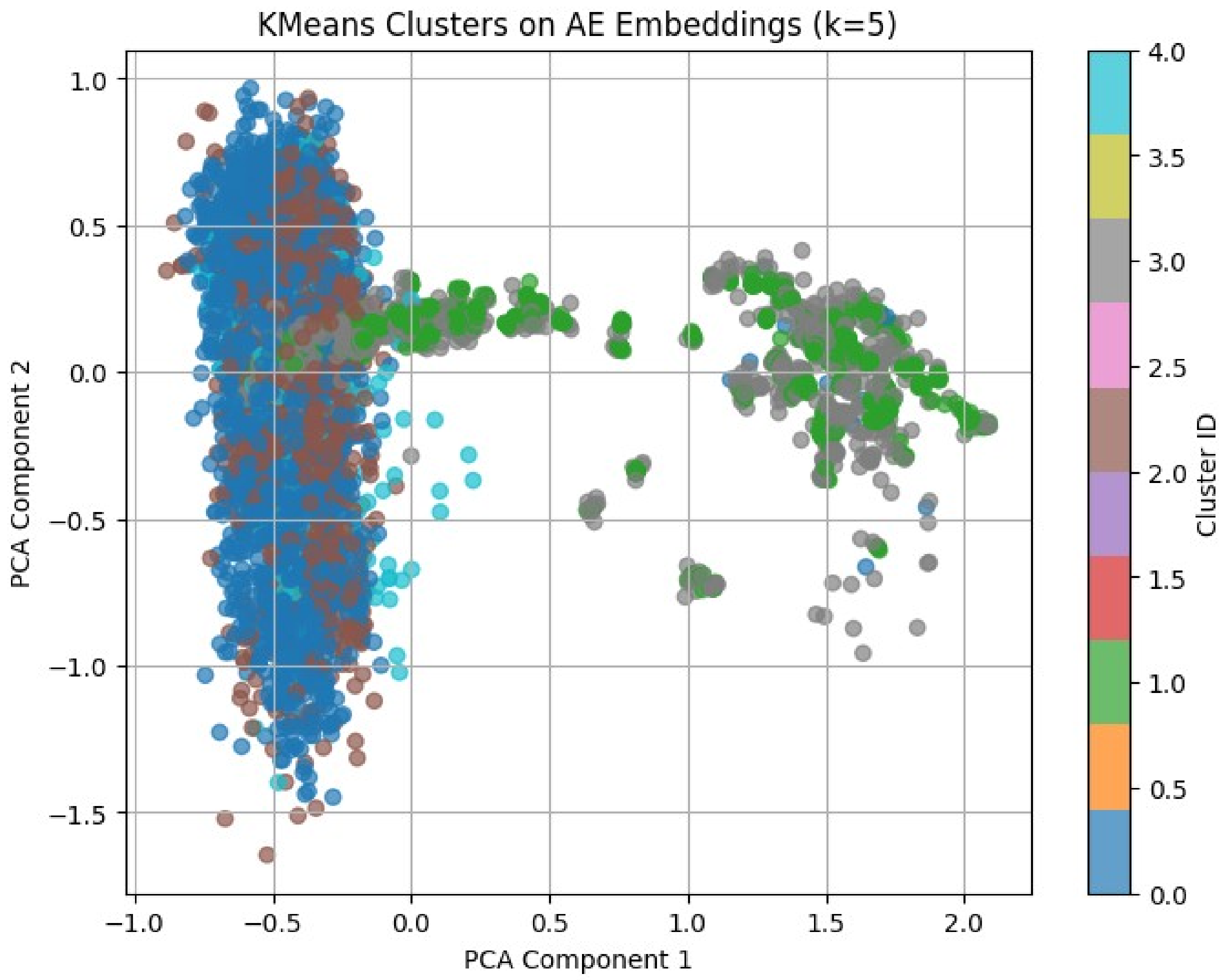
```
ari_scores = []
for k in range(2, 16):
    kmeans = KMeans(n_clusters=k, random_state=42, n_init=10)
    cluster_labels = kmeans.fit_predict(embeddings)
    ari = adjusted_rand_score(train_labels, cluster_labels)
    ari_scores.append(ari)

# Determine best k and best ARI
best_k_index = np.argmax(ari_scores)
best_k = best_k_index + 2
best_ari = ari_scores[best_k_index]
```

```python
# Plotting ARI Scores
plt.figure(figsize=(8, 6))
plt.plot(range(2, 16), ari_scores, marker='o', color='purple')
plt.axvline(x=best_k, color='red', linestyle='--', label=f'Best K =
{best_k}, ARI = {best_ari:.4f}')
plt.xlabel("Number of Clusters (k)")
plt.ylabel("Adjusted Rand Index (ARI)")
plt.title("ARI on Autoencoder Embeddings")
plt.legend()
plt.grid(True)
plt.show()
```



```python
# Choose k based on ARI
best_embedd_ari_k = np.argmax(ari_scores) + 2
# print(f"Best ARI score at k = {best_embedd_ari_k}") # TODO
kmeans_ari = KMeans(n_clusters=best_embedd_ari_k, random_state=42,
n_init=10)
clusters_ari = kmeans_ari.fit_predict(X_train)
```

```
# PCA for visualization
pca =  PCA(n_components=2) # TODO
X_train_pca_ari =  pca.fit_transform(embeddings) # TODO


# Plotting the clusters
plt.figure(figsize=(8, 6))
# TODO <--code below-->
plt.scatter(X_train_pca_ari[:, 0], X_train_pca_ari[:, 1],
c=clusters_ari, cmap='tab10', alpha=0.7)
plt.xlabel("PCA Component 1")
plt.ylabel("PCA Component 2")
plt.title(f"KMeans Clusters on AE Embeddings (k={best_embedd_ari_k})")
plt.colorbar(label="Cluster ID")
plt.grid(True)
plt.show()
```


KMeans Clusters on AE Embeddings (k=5)
PCA Component 1
PCA Component 2
Cluster ID