Appendix

September 2, 2024

For accessing the PhIDDLI model visit the following link:https://github.com/michaeldelves/PhIDDLI/tree/main

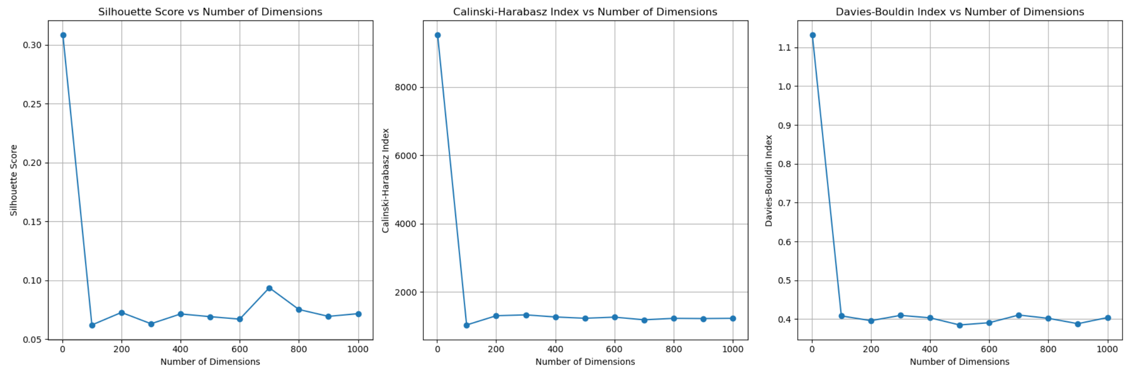
Challenges in High Dimensional Clustering

Observation of Evaluation Metrics over the increase in dimensions

[5]:
```python
import numpy as np
import pandas as pd
from sklearn.neighbors import NearestNeighbors
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score, calinski_harabasz_score,␣
 ↪davies_bouldin_score
import matplotlib.pyplot as plt
import random

# Load the high-dimensional dataset from a CSV file
def load_data_from_csv(file_path):
    data = pd.read_csv(file_path)
    data=data.T
    return data

# Select a subset of dimensions randomly
def select_random_dimensions(data, n_dimensions):
    columns = random.sample(list(data.columns), n_dimensions)
    return data[columns].values

# Compute neighbor occurrence frequency (Hubness)
def compute_hubness(X, k=10):
    nbrs = NearestNeighbors(n_neighbors=k).fit(X)
    distances, indices = nbrs.kneighbors(X)
    neighbor_counts = np.zeros(X.shape[0])
    for idx in indices:
        neighbor_counts[idx] += 1
    return neighbor_counts

# Cluster the dataset and return labels
def cluster_data(X, n_clusters=15):
    kmeans = KMeans(n_clusters=n_clusters, random_state=42)
    labels = kmeans.fit_predict(X)
    return labels

# Main function to run the analysis
def main():
    # Load data from CSV file
    file_path = '/Users/vishanthsuresh/Downloads/Data Science/Dissertation/
 ↪PhIDDLI-main/data/embeddings_aug_B1_primary.csv'  # Replace with your actual␣
 ↪file path
    data = load_data_from_csv(file_path)

    # List of dimensions to iterate over
    dimensions_list = [2] + list(range(100, 1100, 100))

    silhouette_scores = []
    calinski_harabasz_scores = []
    davies_bouldin_scores = []

    for n_dimensions in dimensions_list:
        # Select random dimensions
        X = select_random_dimensions(data, n_dimensions)

        # Compute hubness
        neighbor_counts = compute_hubness(X)

        # Cluster data
        labels = cluster_data(X)

        # Calculate silhouette score
        silhouette = silhouette_score(X, labels)
        silhouette_scores.append(silhouette)

        # Calculate Calinski-Harabasz Index
        calinski_harabasz = calinski_harabasz_score(X, labels)
        calinski_harabasz_scores.append(calinski_harabasz)

        # Calculate Davies-Bouldin Index
        davies_bouldin = 1/davies_bouldin_score(X, labels)
        davies_bouldin_scores.append(davies_bouldin)

        print(f"Dimensions: {n_dimensions}, Silhouette Score: {silhouette},␣
 ↪Calinski-Harabasz Index: {calinski_harabasz}, Davies-Bouldin Index:␣
 ↪{davies_bouldin}")

    # Plot the scores
    plt.figure(figsize=(18, 6))

    # Silhouette Score
    plt.subplot(1, 3, 1)
    plt.plot(dimensions_list, silhouette_scores, marker='o')
    plt.xlabel('Number of Dimensions')
    plt.ylabel('Silhouette Score')
    plt.title('Silhouette Score vs Number of Dimensions')
    plt.grid(True)

    # Calinski-Harabasz Index
    plt.subplot(1, 3, 2)
    plt.plot(dimensions_list, calinski_harabasz_scores, marker='o')
    plt.xlabel('Number of Dimensions')
    plt.ylabel('Calinski-Harabasz Index')
    plt.title('Calinski-Harabasz Index vs Number of Dimensions')
    plt.grid(True)

    # Davies-Bouldin Index
    plt.subplot(1, 3, 3)
    plt.plot(dimensions_list, davies_bouldin_scores, marker='o')
    plt.xlabel('Number of Dimensions')
    plt.ylabel('Davies-Bouldin Index')
    plt.title('Davies-Bouldin Index vs Number of Dimensions')
    plt.grid(True)

    plt.tight_layout()
    plt.show()

if __name__ == "__main__":
    main()
```

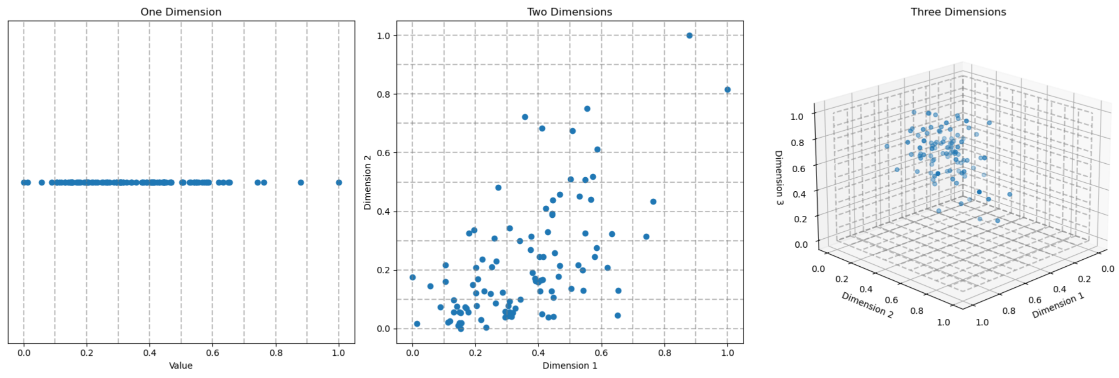
```
Dimensions: 2, Silhouette Score: 0.30821960097749607, Calinski-Harabasz Index:
9521.070155469304, Davies-Bouldin Index: 1.131343338640496
Dimensions: 100, Silhouette Score: 0.061971248215102366, Calinski-Harabasz
Index: 1032.5532715834659, Davies-Bouldin Index: 0.4082109198516956
Dimensions: 200, Silhouette Score: 0.07260070029612507, Calinski-Harabasz Index:
1302.4719463052986, Davies-Bouldin Index: 0.3959735044997108
Dimensions: 300, Silhouette Score: 0.06300499460989602, Calinski-Harabasz Index:
1328.2768845345158, Davies-Bouldin Index: 0.40958132983860834
Dimensions: 400, Silhouette Score: 0.07133252458699957, Calinski-Harabasz Index:
1266.6843030135153, Davies-Bouldin Index: 0.4032594446699761
Dimensions: 500, Silhouette Score: 0.06901342206791014, Calinski-Harabasz Index:
1228.6291928458697, Davies-Bouldin Index: 0.38489365075105264
Dimensions: 600, Silhouette Score: 0.06692588827454864, Calinski-Harabasz Index:
1261.333191801465, Davies-Bouldin Index: 0.39067155735805337
Dimensions: 700, Silhouette Score: 0.09357897841715351, Calinski-Harabasz Index:
1183.7494255766837, Davies-Bouldin Index: 0.4104643810879705
Dimensions: 800, Silhouette Score: 0.07511850105351464, Calinski-Harabasz Index:
1226.3294095765943, Davies-Bouldin Index: 0.4019582877465828
Dimensions: 900, Silhouette Score: 0.06927039780949519, Calinski-Harabasz Index:
```

```
1220.3707033531489, Davies-Bouldin Index: 0.3881637214843754
Dimensions: 1000, Silhouette Score: 0.07156348938036947, Calinski-Harabasz
Index: 1227.3582978178738, Davies-Bouldin Index: 0.40395099656107053
```



[2]:
```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import random

# Load the dataset
df = pd.read_csv('/Users/vishanthsuresh/Downloads/Data Science/Dissertation/
↪PhIDDLI-main/data/embeddings_aug_B1_primary.csv')
df = df.T

# Select 100 random points
random_indices = random.sample(range(len(df)), 100)
random_points = df.iloc[random_indices]

# Normalize the data to range [0, 1]
random_points = (random_points - random_points.min()) / (random_points.max() -␣
↪random_points.min())

# Function to discretize the space and count empty cells
def count_empty_cells(data, dimensions, cell_size):
    grid_shape = tuple(int(1 / cell_size) for _ in range(dimensions))
    grid = np.zeros(grid_shape)

    for point in data:
        indices = tuple(int(coord / cell_size) for coord in point[:dimensions])
        if all(0 <= index < size for index, size in zip(indices, grid_shape)):
            grid[indices] += 1

    empty_cells = np.sum(grid == 0)
    total_cells = grid.size
    return empty_cells, total_cells

# Creating a figure with three subplots in a single row with equal sizes
fig = plt.figure(figsize=(18, 6))  # Increased the height to make the plots␣
 ↪equal in size

# Plotting one dimension as a straight line with grid cells
ax1 = fig.add_subplot(131)
ax1.plot(random_points.iloc[:, 0], np.zeros(100), 'o')
ax1.set_title('One Dimension')
ax1.set_xlabel('Value')
ax1.set_yticks([])  # Hide y-axis
# Add grid lines
for x in np.arange(0, 1.1, 0.1):
    ax1.axvline(x, color='gray', linestyle='--', alpha=0.5)

# Plotting two dimensions with grid cells
ax2 = fig.add_subplot(132)
ax2.scatter(random_points.iloc[:, 0], random_points.iloc[:, 1])
ax2.set_title('Two Dimensions')
ax2.set_xlabel('Dimension 1')
ax2.set_ylabel('Dimension 2')
# Add grid lines
for x in np.arange(0, 1.1, 0.1):
    ax2.axvline(x, color='gray', linestyle='--', alpha=0.5)
    ax2.axhline(x, color='gray', linestyle='--', alpha=0.5)

# Plotting three dimensions with grid cells
ax3 = fig.add_subplot(133, projection='3d')
ax3.scatter(random_points.iloc[:, 0], random_points.iloc[:, 1], random_points.
 ↪iloc[:, 2])
ax3.set_title('Three Dimensions')
ax3.set_xlabel('Dimension 1')
ax3.set_ylabel('Dimension 2')
ax3.set_zlabel('Dimension 3')
ax3.view_init(elev=20., azim=45)  # Adjust the angle for better visualization
# Add grid lines
for x in np.arange(0, 1.1, 0.1):
    ax3.plot([x, x], [0, 0], [0, 1], color='gray', linestyle='--', alpha=0.5)
    ax3.plot([x, x], [0, 1], [0, 0], color='gray', linestyle='--', alpha=0.5)
    ax3.plot([0, 0], [x, x], [0, 1], color='gray', linestyle='--', alpha=0.5)
    ax3.plot([0, 1], [x, x], [0, 0], color='gray', linestyle='--', alpha=0.5)
    ax3.plot([0, 0], [0, 1], [x, x], color='gray', linestyle='--', alpha=0.5)
    ax3.plot([0, 1], [0, 0], [x, x], color='gray', linestyle='--', alpha=0.5)

plt.tight_layout()
plt.show()
```

```
# Discretizing the space and counting empty cells
cell_size = 0.1

# One dimension
empty_cells_1d, total_cells_1d = count_empty_cells(random_points.values, 1,␣
 ↪cell_size)
print(f"One Dimension: {empty_cells_1d} empty cells out of {total_cells_1d}")

# Two dimensions
empty_cells_2d, total_cells_2d = count_empty_cells(random_points.values, 2,␣
 ↪cell_size)
print(f"Two Dimensions: {empty_cells_2d} empty cells out of {total_cells_2d}")

# Three dimensions
empty_cells_3d, total_cells_3d = count_empty_cells(random_points.values, 3,␣
 ↪cell_size)
print(f"Three Dimensions: {empty_cells_3d} empty cells out of {total_cells_3d}")
```



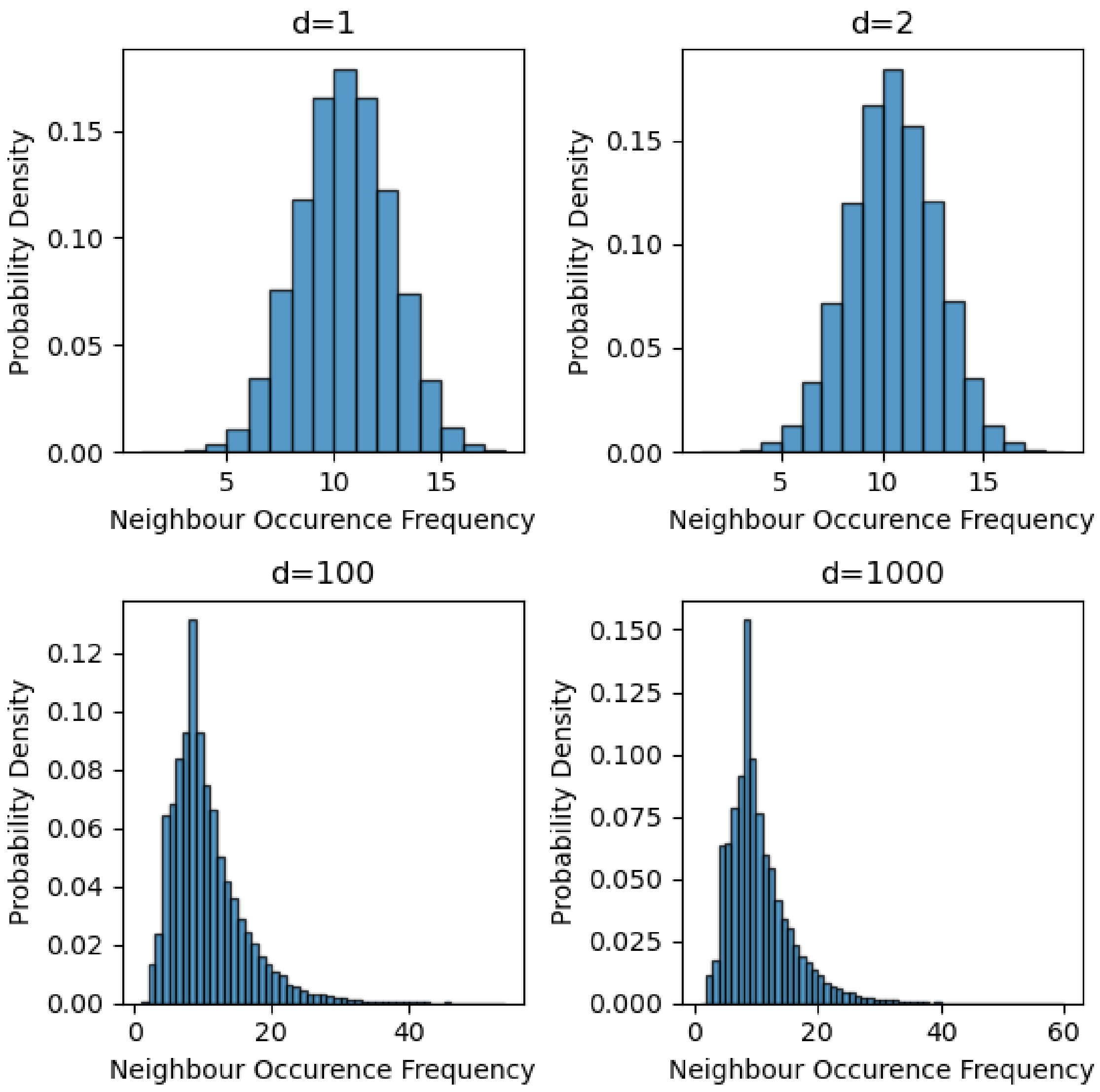
```
One Dimension: 1 empty cells out of 10
Two Dimensions: 65 empty cells out of 100
Three Dimensions: 928 empty cells out of 1000
```

1 Hubness effect

[57]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neighbors import NearestNeighbors

# Load the CSV dataset
dataset = pd.read_csv('/Users/vishanthsuresh/Downloads/Data Science/Dissertation/
 ↪PhIDDLI-main/data/embeddings_aug_B1_primary.csv')
```

```
dataset=dataset.T
# Function to calculate neighbor occurrence frequency
def neighbor_occurrence_frequency(data, k=10):
    nbrs = NearestNeighbors(n_neighbors=k).fit(data)
    _, indices = nbrs.kneighbors(data)
    occurrence_counts = np.zeros(len(data))

    for neighbors in indices:
        for neighbor in neighbors:
            occurrence_counts[neighbor] += 1

    return occurrence_counts

# Function to plot neighbor occurrence frequency distribution
def plot_distribution(ax, data, dimensions, k=10):
    occurrence_counts = neighbor_occurrence_frequency(data.iloc[:, :dimensions],␣
 ↪k)
    ax.hist(occurrence_counts, bins=range(1, max(occurrence_counts.astype(int))␣
 ↪+ 2), density=True, alpha=0.75, edgecolor='black')
    ax.set_title(f'd={dimensions}')
    ax.set_xlabel('Neighbour Occurence Frequency')
    ax.set_ylabel('Probability Density')

# Create a 2x2 grid of subplots
fig, axes = plt.subplots(2, 2, figsize=(6, 6))

# Single dimension
plot_distribution(axes[0, 0], dataset, dimensions=1)

# Two dimensions
plot_distribution(axes[0, 1], dataset, dimensions=2)

# 100 dimensions (assuming the dataset has at least 100 dimensions)
if dataset.shape[1] >= 100:
    plot_distribution(axes[1, 0], dataset, dimensions=100)
else:
    axes[1, 0].text(0.5, 0.5, 'Not enough dimensions',␣
 ↪horizontalalignment='center', verticalalignment='center', transform=axes[1, 0].
 ↪transAxes)

# 1000 dimensions (assuming the dataset has at least 1000 dimensions)
if dataset.shape[1] >= 1000:
    plot_distribution(axes[1, 1], dataset, dimensions=1000)
else:
    axes[1, 1].text(0.5, 0.5, 'Not enough dimensions',␣
 ↪horizontalalignment='center', verticalalignment='center', transform=axes[1, 1].
 ↪transAxes)
```

```
# Adjust layout and show the plots
plt.tight_layout()
plt.show()
```



[64]:
```
import numpy as np
import pandas as pd
from sklearn.neighbors import NearestNeighbors
from scipy.stats import skew

# Load the CSV dataset
```

```
dataset = pd.read_csv('/Users/vishanthsuresh/Downloads/Data Science/Dissertation/
↪PhIDDLI-main/data/embeddings_aug_B1_primary.csv')
dataset=dataset.T
# Function to calculate neighbor occurrence frequency
def neighbor_occurrence_frequency(data, k=10):
    nbrs = NearestNeighbors(n_neighbors=k).fit(data)
    _, indices = nbrs.kneighbors(data)
    occurrence_counts = np.zeros(len(data))

    for neighbors in indices:
        for neighbor in neighbors:
            occurrence_counts[neighbor] += 1

    return occurrence_counts

# Function to calculate skewness of the neighbor occurrence frequency␣
↪distribution
def calculate_skewness(data, dimensions, k=10):
    data_subset = data.iloc[:, :dimensions]
    occurrence_counts = neighbor_occurrence_frequency(data_subset, k)
    skewness_value = skew(occurrence_counts)
    return skewness_value

# Function to print skewness for given dimensions
def print_skewness_for_dimensions(dataset, dimensions, k=10):
    for dim in dimensions:
        if dataset.shape[1] >= dim:
            skewness_value = calculate_skewness(dataset, dim, k)
            print(f'Skewness for {dim} dimensions: {skewness_value}')
        else:
            print(f'The dataset does not have {dim} dimensions.')

# Specify the dimensions to be used
dimensions_to_test = [1, 2, 100, 1000]

# Print skewness for specified dimensions
print_skewness_for_dimensions(dataset, dimensions_to_test)
```

```
Skewness for 1 dimensions: -0.009939694892076769
Skewness for 2 dimensions: -0.019795075849741002
Skewness for 100 dimensions: 1.6608874957623627
Skewness for 1000 dimensions: 1.6948153769840806
```

2 Overall Result

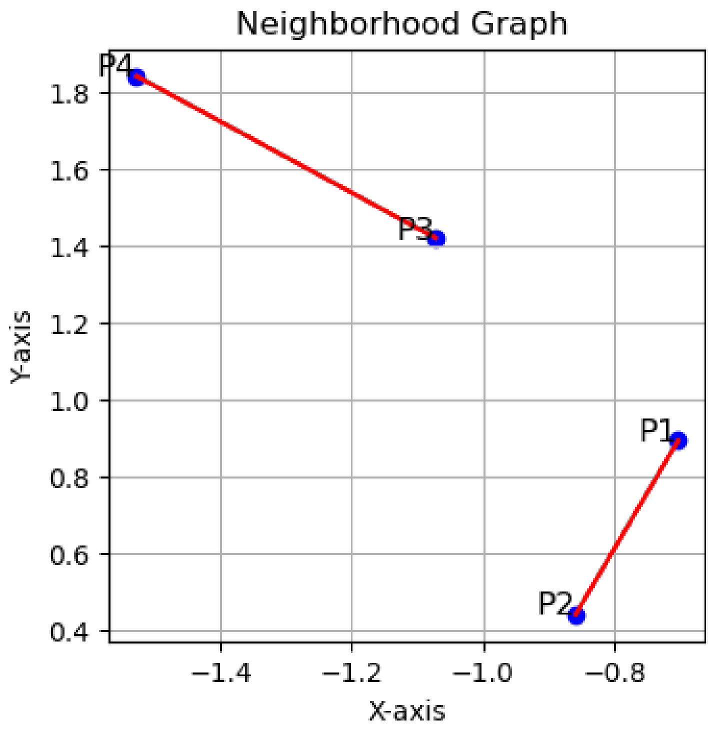
From these experiments, reducing the dimensions can be proved as a way for better clustering

Isomap

September 2, 2024

0.0.1 Isomap Sample Calculation and Application

[24]:
```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial import distance_matrix

# Given points
points = np.array([
    [-0.70323211,  0.89485216],
    [-0.85881805,  0.4431116 ],
    [-1.07164979,  1.42083216],
    [-1.52713239,  1.84265852]
])

# Calculate the distance matrix
distances = distance_matrix(points, points)

# Create a neighborhood graph where each point is connected to its nearest␣
 ↪neighbor
plt.figure(figsize=(4, 4))

# Plot the points
plt.scatter(points[:, 0], points[:, 1], color='blue')
for i, (x, y) in enumerate(points):
    plt.text(x, y, f'P{i+1}', fontsize=12, ha='right')

# Find and plot connections to nearest neighbors
for i in range(len(points)):
    # Ignore the distance to itself by setting it to infinity
    distances[i, i] = np.inf
    # Find the index of the nearest neighbor
    nearest_neighbor_idx = np.argmin(distances[i])
    # Plot the edge to the nearest neighbor
    plt.plot([points[i, 0], points[nearest_neighbor_idx, 0]],
             [points[i, 1], points[nearest_neighbor_idx, 1]], 'r-')

# Set plot title and labels
plt.title('Neighborhood Graph ')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.grid(True)
plt.show()
```



[31]:
```
import numpy as np

# Define the matrix B
B = np.array([
    [-0.125388,  0.125388, -24.76043,  24.76043],
    [ 0.125388, -0.125388,  24.76043, -24.76043],
    [-24.76043,  24.76043, -0.125388,  0.125388],
    [ 24.76043, -24.76043,  0.125388, -0.125388]
])

# Perform eigen decomposition
eigenvalues, eigenvectors = np.linalg.eig(B)

# Sort the eigenvalues and eigenvectors by the absolute value of eigenvalues in␣
 ↪descending order
sorted_indices = np.argsort(-np.abs(eigenvalues))
eigenvalues = eigenvalues[sorted_indices]
eigenvectors = eigenvectors[:, sorted_indices]
```

```
# Select the principal eigenvector (corresponding to the largest eigenvalue)
principal_eigenvector = eigenvectors[:, 0]

# Normalize the principal eigenvector (if needed)
principal_eigenvector = principal_eigenvector / np.linalg.
 ↪norm(principal_eigenvector)

# The 1D embeddings are the entries of the normalized principal eigenvector
embedding_1D = principal_eigenvector

# Print the 1D embeddings
print("1D Embeddings:", embedding_1D)
```

```
1D Embeddings: [-0.5  0.5 -0.5  0.5]
```

[ ]:
```
import pickle
from argparse import ArgumentParser
from pathlib import Path
from typing import List, Tuple
import time

import numpy as np
from loguru import logger
from tqdm import tqdm
from sklearn.manifold import Isomap

# Function to apply ISOMAP transformation
def apply_isomap(vectors: np.ndarray, n_neighbors: int = 1, n_components: int =␣
 ↪2) -> np.ndarray:
    logger.info(f"Applying ISOMAP with {n_neighbors} neighbors and␣
 ↪{n_components} components")
    start_time = time.time()
    isomap = Isomap(n_neighbors=n_neighbors, n_components=n_components)
    result = isomap.fit_transform(vectors)
    end_time = time.time()
    time_taken = end_time - start_time
    logger.info(f"ISOMAP took {time_taken:.2f} seconds")
    return result

def load_embeddings(file: Path) -> Tuple[List[Path], np.ndarray]:
    embeddings = pickle.loads(file.read_bytes())
    files, vectors = zip(*list(embeddings.items()))
    vectors = np.array(vectors)  # Convert the list of vectors to a numpy array
    return files, vectors
```

```
def reduce_dimensionality(embeddings_file: Path, output_file: Path, n_neighbors:␣
↪int = 15, n_components: int = 2):
    files, vectors = load_embeddings(file=embeddings_file)
    points = apply_isomap(vectors=vectors, n_neighbors=n_neighbors,␣
↪n_components=n_components)
    points_mapping = dict(zip(files, points))
    output_file.write_bytes(pickle.dumps(points_mapping))

def set_paths_and_run(embeddings_file: str, output_file: str, n_neighbors: int =␣
↪15, n_components: int = 2):
    reduce_dimensionality(
        embeddings_file=Path(embeddings_file),
        output_file=Path(output_file),
        n_neighbors=n_neighbors,
        n_components=n_components
    )

if __name__ == "__main__":
    # Example programmatic usage
    embeddings_file = "/Users/vishanthsuresh/Downloads/Data Science/Dissertation/
↪PhIDDLI-main/data/embeddings_aug_B1_primary.pkl"
    output_file = "/Users/vishanthsuresh/Downloads/Data Science/Dissertation/
↪PhIDDLI-main/data/reduced_isomap_1.pkl"

    set_paths_and_run(embeddings_file, output_file)

    # Uncomment the following lines if you want to use command-line arguments␣
↪instead
    # parser = ArgumentParser()
    # parser.add_argument('embeddings_file', type=Path)
    # parser.add_argument('--output-file', type=Path, required=True)
    # parser.add_argument('--n_neighbors', type=int, default=5, help='Number of␣
↪neighbors to consider for each point')
    # parser.add_argument('--n_components', type=int, default=2, help='Number of␣
↪dimensions to reduce to')
    # args = parser.parse_args()
    # reduce_dimensionality(
    #     embeddings_file=args.embeddings_file,
    #     output_file=args.output_file,
    #     n_neighbors=args.n_neighbors,
    #     n_components=args.n_components)
```

```
2024-07-30 20:50:08.987 | INFO     |
__main__:apply_isomap:14 - Applying ISOMAP with

15 neighbors and 2 components
```

```
[ ]:
```

UMAP

September 2, 2024

0.0.1 UMAP Sample Calculation

[1]:
```python
import numpy as np
from scipy.spatial.distance import pdist, squareform
import umap

# Step 1: Define the input data
points = np.array([
    [-0.70323211, 0.89485216],
    [-0.85881805, 0.4431116],
    [-1.07164979, 1.42083216],
    [-1.52713239, 1.84265852]
])

# Step 2: Calculate pairwise Euclidean distances
pairwise_distances = squareform(pdist(points, metric='euclidean'))
print("Pairwise Distance Matrix:")
print(pairwise_distances)
```

```
Pairwise Distance Matrix:
[[0.         0.47778292 0.6421733  1.25584576]
 [0.47778292 0.         1.00061723 1.55092735]
 [0.6421733  1.00061723 0.         0.62080744]
 [1.25584576 1.55092735 0.62080744 0.        ]]
```

[5]:
```python
# Create a copy of the pairwise distance matrix to modify
pairwise_distances_no_self = np.copy(pairwise_distances)

# Set the diagonal to a very large number to effectively ignore self-distances
np.fill_diagonal(pairwise_distances_no_self, np.inf)

# Calculate the local radius (minimum non-self distance for each point)
local_radius = np.min(pairwise_distances_no_self, axis=1)

print("\nLocal Radius (ρ):")
print(local_radius)
```

```

Local Radius (ρ):
```

```
[0.47778292 0.47778292 0.62080744 0.62080744]
```

[6]:
```
# Step 4: Compute Fuzzy Set Membership Strength (μ)
def compute_fuzzy_membership(distances, rho):
    sigma = 1.0  # This is a scaling factor; in practice, this would be␣
 ↪optimized or chosen based on data
    return np.exp(-(distances - rho[:, np.newaxis]) / sigma)

fuzzy_membership = compute_fuzzy_membership(pairwise_distances, local_radius)
print("\nFuzzy Membership Strength (μ):")
print(fuzzy_membership)
```

```
Fuzzy Membership Strength (μ):
[[1.6124954  1.         0.84841075 0.45929488]
 [1.         1.6124954  0.59283788 0.34193165]
 [0.97886077 0.6839915  1.86042962 1.        ]
 [0.52991518 0.3945064  1.         1.86042962]]
```

[7]:
```
# Step 5: Symmetric Fuzzy Set Construction
symmetric_weights = fuzzy_membership + fuzzy_membership.T - fuzzy_membership *␣
 ↪fuzzy_membership.T
print("\nSymmetric Fuzzy Set Weights (w):")
print(symmetric_weights)
```

```
Symmetric Fuzzy Set Weights (w):
[[0.62484938 1.         0.99679552 0.74582273]
 [1.         0.62484938 0.87133331 0.60154383]
 [0.99679552 0.87133331 0.25966087 1.        ]
 [0.74582273 0.60154383 1.         0.25966087]]
```

[8]:
```
# Step 6: Applying UMAP for Dimensionality Reduction
# Initialize UMAP with n_components=1 for 1D projection
umap_reducer = umap.UMAP(n_components=1, random_state=42)

# Fit and transform the data
points_1d = umap_reducer.fit_transform(points)

print("\nReduced 1D Coordinates:")
print(points_1d)
```

```
/Users/vishanthsuresh/anaconda3/lib/python3.11/site-packages/umap/umap_.py:1945:
UserWarning: n_jobs value 1 overridden to 1 by setting random_state. Use no seed
for parallelism.
  warn(f"n_jobs value {self.n_jobs} overridden to 1 by setting random_state. Use
no seed for parallelism.")
/Users/vishanthsuresh/anaconda3/lib/python3.11/site-packages/umap/umap_.py:2437:
UserWarning: n_neighbors is larger than the dataset size; truncating to
```

```
X.shape[0] - 1
  warn(


Reduced 1D Coordinates:
[[32.97621 ]
 [32.166805]
 [30.985752]
 [30.07776 ]]
```

Clustering_Umap_experiments

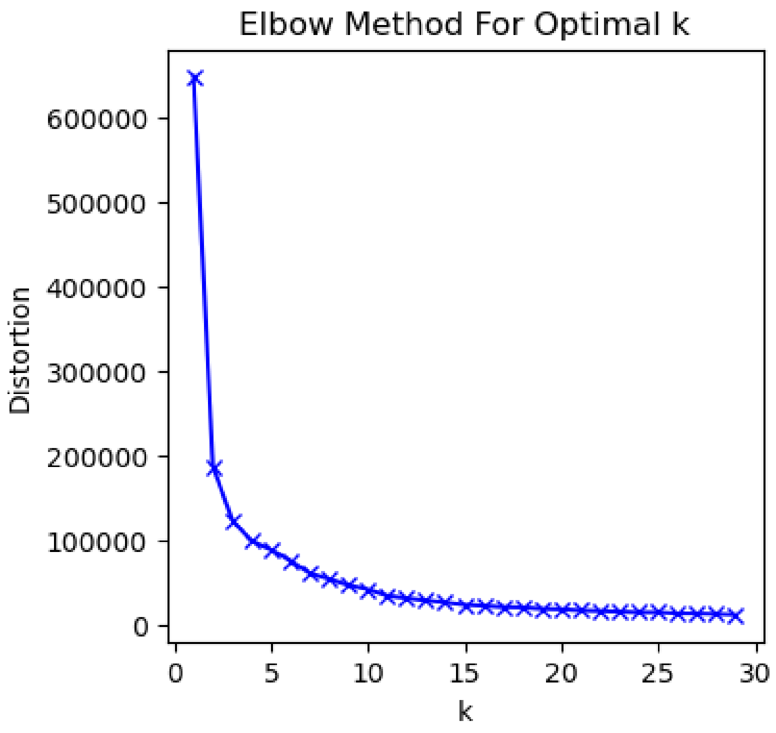
September 2, 2024

1 Model with UMAP and Kmeans

The low dimensional data points are processed for clustering as evaluating high dimensional data seems problematic.

[62]:
```
import pickle
from argparse import ArgumentParser
from pathlib import Path
from typing import List, Optional, Tuple

import numpy as np
from loguru import logger
from tqdm import tqdm
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
from kneed import KneeLocator

def load_embeddings(file: Path) -> Tuple[List[Path], np.ndarray]:
    embeddings = pickle.loads(file.read_bytes())
    files, vectors = zip(*list(embeddings.items()))
    vectors = np.array(vectors)  # Ensure vectors are a NumPy array
    return files, vectors


def make_clusters(vectors: np.ndarray, n_clusters: int) -> np.ndarray:
    logger.info(f"Performing KMeans clustering with {n_clusters} clusters")
    kmeans = KMeans(n_clusters=n_clusters, random_state=42)
    clusters = kmeans.fit_predict(vectors)
    return clusters


def find_optimal_number_of_clusters(data: np.ndarray) -> int:
    logger.info("Finding optimal value for k ...")
    distortions = []
    K = range(1, 30)
    for k in K:
        kmeanModel = KMeans(n_clusters=k)
        kmeanModel.fit(data)
        distortions.append(kmeanModel.inertia_)

    # Plotting the elbow curve
    plt.figure(figsize=(4,4))
    plt.plot(K, distortions, 'bx-')
    plt.xlabel('k')
    plt.ylabel('Distortion')
    plt.title('Elbow Method For Optimal k')
    plt.show()

    kneedle = KneeLocator(K, distortions, direction='decreasing', curve='convex')
    logger.info(f"Elbow point at k={kneedle.elbow}")
    #return kneedle.elbow if kneedle.elbow else 10# Default to 10 if elbow not␣
 ↪found
    return 4


def compute_clusters(embeddings_file: Path, output_file: Path, clusters:␣
 ↪Optional[int] = None):
    files, vectors = load_embeddings(file=embeddings_file)
    if clusters is None:
        clusters = find_optimal_number_of_clusters(data=vectors)
    clusters = make_clusters(vectors=vectors, n_clusters=clusters)
    cluster_mapping = dict(zip(files, clusters))
    output_file.write_bytes(pickle.dumps(cluster_mapping))


def set_paths_and_run(embeddings_file: str, output_file: str, clusters:␣
 ↪Optional[int] = None):
    compute_clusters(
        embeddings_file=Path(embeddings_file),
        output_file=Path(output_file),
        clusters=clusters
    )


if __name__ == "__main__":
    # Example programmatic usage
    embeddings_file = "/Users/vishanthsuresh/Downloads/Data Science/Dissertation/
 ↪PhIDDLI-main/data/reduced_UMAP.pkl"
    output_file = "/Users/vishanthsuresh/Downloads/Data Science/Dissertation/
 ↪PhIDDLI-main/data/rKMeans_clusters_new.pkl"
    clusters = None  # Or specify an integer for a fixed number of clusters

    set_paths_and_run(embeddings_file, output_file, clusters)
    # Uncomment the following lines if you want to use command-line arguments␣
↪instead
    # parser = ArgumentParser()
    # parser.add_argument('embeddings_file', type=Path)
    # parser.add_argument('--clusters', type=lambda x: int(x) if x else None,␣
↪default=None, required=False)
    # parser.add_argument('--output-file', type=Path, required=True)
    # args = parser.parse_args()
    # compute_clusters(
    #     embeddings_file=args.embeddings_file,
    #     output_file=args.output_file,
    #     clusters=args.clusters)
```

```
2024-08-09 01:42:04.434 | INFO     |
__main__:find_optimal_number_of_clusters:28 -
Finding optimal value for k ...
```



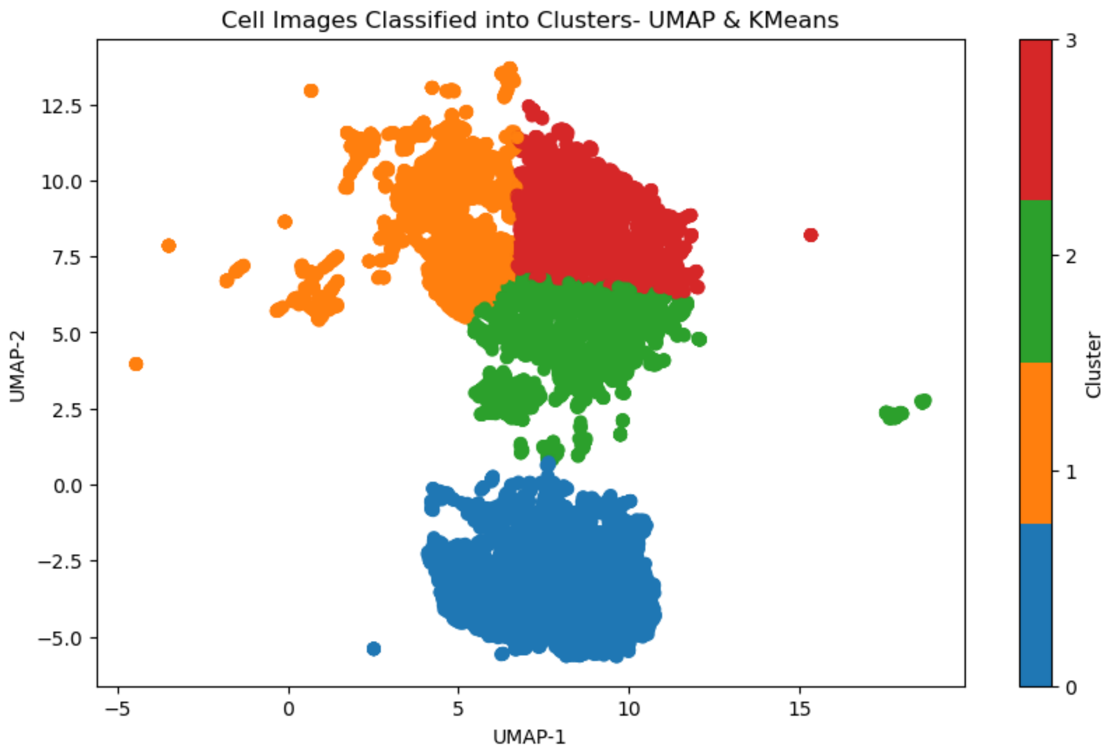
```
2024-08-09 01:42:05.055 | INFO     |
__main__:find_optimal_number_of_clusters:45 -
Elbow point at k=4
2024-08-09 01:42:05.056 | INFO     |
__main__:make_clusters:21 - Performing KMeans

clustering with 4 clusters
```

As we can see here that the approach for finding the optimal k value has been changed. In the old PhiDDLI approach, the each k value is tested after getting increamented by 5, whereas here k value is tested by increamenting one by one.

[63]:
```python
import pickle
from argparse import ArgumentParser
from collections import defaultdict
from pathlib import Path
from typing import List, Tuple

import pandas as pd

def load_file_contents(file: Path) -> Tuple[List[Path], List]:
    return pickle.loads(file.read_bytes())

def export_csv(clusters_file: Path, points_file: Path, output_file: Path):
    clusters_data = load_file_contents(clusters_file)
    points_data = load_file_contents(points_file)

    data = defaultdict(dict)
    for key, cluster in clusters_data.items():
        data[key]["cluster"] = cluster
        point = points_data[key]
        data[key]["point_x"], data[key]["point_y"] = point

    df: pd.DataFrame = pd.DataFrame.from_dict(data, orient="index") \
        .rename_axis('cell_location') \
        .reset_index() \
        .assign(cell_location=lambda df: df.cell_location.apply(Path)) \
        .assign(parent_image=lambda df: df.cell_location.apply(lambda path: path.
↪parent.name))

    df.to_csv(output_file, index=False)

def set_paths_and_run(clusters_file: str, points_file: str, output_file: str):
    export_csv(
        clusters_file=Path(clusters_file),
        points_file=Path(points_file),
        output_file=Path(output_file)
    )

if __name__ == "__main__":
    # Example programmatic usage
    clusters_file = "/Users/vishanthsuresh/Downloads/Data Science/Dissertation/
↪PhIDDLI-main/data/rKMeans_clusters_new.pkl"
    points_file = "/Users/vishanthsuresh/Downloads/Data Science/Dissertation/
↪PhIDDLI-main/data/reduced_UMAP.pkl"
```

```
    output_file = "/Users/vishanthsuresh/Downloads/Data Science/Dissertation/
↪PhIDDLI-main/data/routput_UMAP_Kmeans_new.csv"

    set_paths_and_run(clusters_file, points_file, output_file)

    # Uncomment the following lines if you want to use command-line arguments␣
↪instead
    # parser = ArgumentParser()
    # parser.add_argument('--clusters', type=Path, required=True)
    # parser.add_argument('--points', type=Path, required=True)
    # parser.add_argument('--output-file', type=Path, required=True)
    # args = parser.parse_args()
    # export_csv(
    #     clusters_file=args.clusters,
    #     points_file=args.points,
    #     output_file=args.output_file)
```

[4]:
```
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.colors as mcolors


data= pd.read_csv("/Users/vishanthsuresh/Downloads/Data Science/Dissertation/
↪PhIDDLI-main/data/routput_UMAP_Kmeans_new.csv")
# Define a set of more contrasting and visually distinct colors
contrasting_colors = [
    '#1f77b4', '#ff7f0e', '#2ca02c','#d62728'
]

# Create a colormap from the list of contrasting colors
custom_cmap_contrasting = mcolors.ListedColormap(contrasting_colors[:15])

# Plotting point_x and point_y with clusters in different colors using the␣
↪custom colormap
plt.figure(figsize=(10, 6))

# Using a scatter plot to visualize the clusters
scatter = plt.scatter(data['point_x'], data['point_y'], c=data['cluster'],␣
↪cmap=custom_cmap_contrasting)

# Adding a color bar to indicate the cluster colors
cbar = plt.colorbar(scatter, label='Cluster', ticks=range(15))
cbar.set_ticklabels(range(15))

# Adding titles and labels
plt.title('Cell Images Classified into Clusters- UMAP & KMeans')
plt.xlabel('UMAP-1')
plt.ylabel('UMAP-2')

# Display the plot
plt.show()
```



[65]:
```
import pandas as pd
from sklearn.metrics import silhouette_score, calinski_harabasz_score,␣
↪davies_bouldin_score

# Load the CSV file
file_path = '/Users/vishanthsuresh/Downloads/Data Science/Dissertation/
↪PhIDDLI-main/data/routput_UMAP_Kmeans_new.csv'
data = pd.read_csv(file_path)

# Extracting the relevant data for clustering validation
vectors = data[['point_x', 'point_y']].values
labels = data['cluster'].values

# Calculate the validation metrics
silhouette_avg = silhouette_score(vectors, labels)
calinski_harabasz_avg = calinski_harabasz_score(vectors, labels)
davies_bouldin_avg = davies_bouldin_score(vectors, labels)
```

```
print(f"Silhouette Score: {silhouette_avg}")
print(f"Calinski-Harabasz Index: {calinski_harabasz_avg}")
print(f"Davies-Bouldin Index: {davies_bouldin_avg}")
```

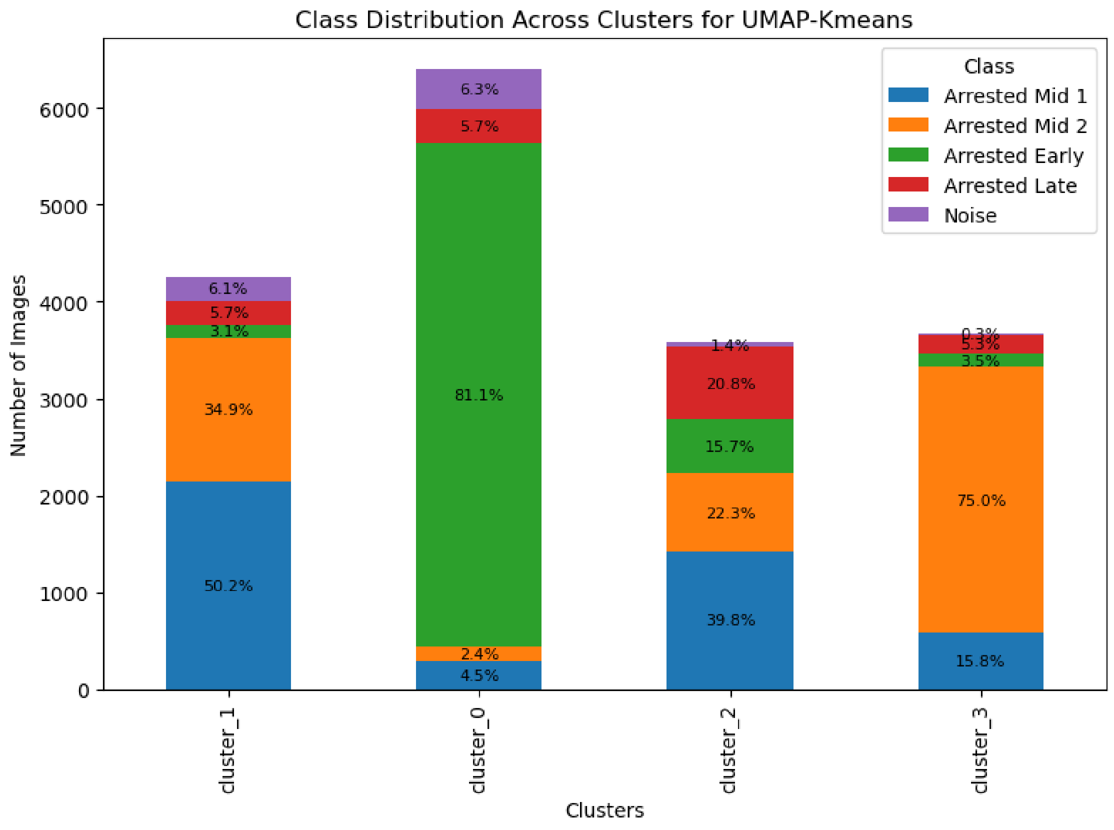
```
Silhouette Score: 0.46425583977749024
Calinski-Harabasz Index: 33310.15230476035
Davies-Bouldin Index: 0.8345935422076254
```

[3]:
```
import os
import torch
import torch.nn as nn
from torchvision import transforms
from PIL import Image
import pandas as pd
import matplotlib.pyplot as plt

# Define CNN Model with three hidden layers (same as before)
class CustomCNN(nn.Module):
    def __init__(self):
        super(CustomCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.maxpool1 = nn.MaxPool2d(kernel_size=2)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.conv4 = nn.Conv2d(128, 128, kernel_size=3, padding=1)
        self.maxpool2 = nn.MaxPool2d(kernel_size=2)
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(128 * 8 * 8, 512)
        self.fc2 = nn.Linear(512, 256)  # Third hidden layer
        self.fc3 = nn.Linear(256, 128)  # Fourth hidden layer
        self.fc4 = nn.Linear(128, 30)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.maxpool1(self.relu(self.conv1(x)))
        x = self.maxpool1(self.relu(self.conv2(x)))
        x = self.relu(self.conv3(x))
        x = self.maxpool2(self.relu(self.conv4(x)))
        x = self.flatten(x)
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))  # Third hidden layer
        x = self.relu(self.fc3(x))  # Fourth hidden layer
        x = self.fc4(x)
        return x

# Load the checkpoint
```

```
checkpoint = torch.load('/Users/vishanthsuresh/Downloads/Data Science/
↪Dissertation/CNN Model/final_model_checkpoint.pth')
model = CustomCNN()
model.load_state_dict(checkpoint['model_state_dict'])
model.eval()

# Define the transformation for the images
transform = transforms.Compose([
    transforms.Resize((64, 64)),
    transforms.ToTensor(),
])

# Define a dictionary to map class indices to class names
class_names = {
    0: 'Arrested Mid 1',
    1: 'Arrested Mid 2',
    2: 'Arrested Early',
    3: 'Arrested Late',
    4: 'Noise'
    # Add other class mappings as necessary
}

# Function to predict image
def predict_image(image_path, model, transform):
    image = Image.open(image_path)
    image = transform(image)
    image = image.unsqueeze(0)  # Add batch dimension
    with torch.no_grad():
        output = model(image)
    _, predicted = torch.max(output, 1)
    return class_names[predicted.item()]

# Directory containing all the clusters
base_folder = '/Users/vishanthsuresh/Downloads/Data Science/Dissertation/
↪PhIDDLI-main/data/clusters_umap_kmeans'
# Initialize an empty DataFrame with columns for each class
all_class_counts = pd.DataFrame(columns=class_names.values())
cluster_names = []

# Iterate over all subfolders and predict images
for folder_name in os.listdir(base_folder):
    folder_path = os.path.join(base_folder, folder_name)
    if os.path.isdir(folder_path):  # Ensure it's a directory
        results = []
        for image_name in os.listdir(folder_path):
            if image_name.endswith(('.png', '.jpg', '.jpeg')):  # Add other␣
↪image extensions if needed
```

```
                image_path = os.path.join(folder_path, image_name)
                predicted_class = predict_image(image_path, model, transform)
                results.append({'Image': image_name, 'Predicted_Class':␣
↪predicted_class})

        # Save results to CSV
        results_df = pd.DataFrame(results)
        results_csv_path = os.path.join(folder_path, f'{folder_name}_predictions.
↪csv')
        results_df.to_csv(results_csv_path, index=False)
        print(f"Results saved to {results_csv_path}")

        # Store class counts for the current cluster
        class_counts = results_df['Predicted_Class'].value_counts()
        # Add missing classes with 0 count
        class_counts = class_counts.reindex(all_class_counts.columns,␣
↪fill_value=0)
        all_class_counts = pd.concat([all_class_counts, class_counts.to_frame().
↪T], ignore_index=True)
        cluster_names.append(folder_name)

# Set index to cluster names
all_class_counts.index = cluster_names

# Plot stacked bar chart with percentage annotations
fig, ax = plt.subplots(figsize=(8, 6))
bars = all_class_counts.plot(kind='bar', stacked=True, ax=ax)

# Calculate percentages and annotate
for i, cluster in enumerate(all_class_counts.index):
    total = all_class_counts.loc[cluster].sum()
    cumulative_sum = 0
    for j, class_name in enumerate(all_class_counts.columns):
        class_count = all_class_counts.loc[cluster, class_name]
        cumulative_sum += class_count
        if class_count > 0:
            percentage = (class_count / total) * 100
            ax.text(i, cumulative_sum - class_count / 2, f'{percentage:.1f}%',␣
↪ha='center', va='center', fontsize=8)

plt.title('Class Distribution Across Clusters for UMAP-Kmeans')
plt.xlabel('Clusters')
plt.ylabel('Number of Images')
plt.legend(title='Class')
plt.tight_layout()
plt.show()
```

```
Results saved to /Users/vishanthsuresh/Downloads/Data
Science/Dissertation/PhIDDLI-
main/data/clusters_umap_kmeans/cluster_1/cluster_1_predictions.csv
Results saved to /Users/vishanthsuresh/Downloads/Data
Science/Dissertation/PhIDDLI-
main/data/clusters_umap_kmeans/cluster_0/cluster_0_predictions.csv
Results saved to /Users/vishanthsuresh/Downloads/Data
Science/Dissertation/PhIDDLI-
main/data/clusters_umap_kmeans/cluster_2/cluster_2_predictions.csv
Results saved to /Users/vishanthsuresh/Downloads/Data
Science/Dissertation/PhIDDLI-
main/data/clusters_umap_kmeans/cluster_3/cluster_3_predictions.csv
```



The images has been clustered and segregated into separate folders for analysis.

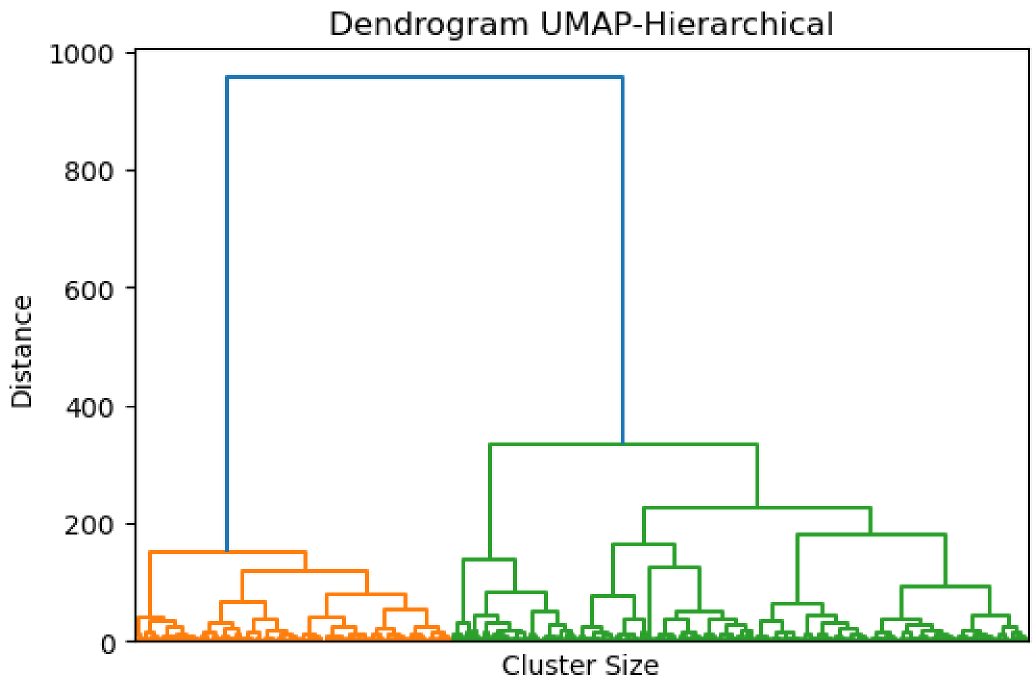
2 UMAP and Hierarchical Clustering

[37]:
```
import pickle
from pathlib import Path
from typing import List, Tuple
```

```
import numpy as np
from loguru import logger
from scipy.cluster.hierarchy import linkage, dendrogram
import matplotlib.pyplot as plt

def load_embeddings(file: Path) -> Tuple[List[Path], np.ndarray]:
    with file.open('rb') as f:
        embeddings = pickle.load(f)
    files, vectors = zip(*list(embeddings.items()))
    vectors = np.array(vectors)  # Ensure vectors are a NumPy array
    return files, vectors

def plot_dendrogram(vectors: np.ndarray, output_file: Path):
    Z = linkage(vectors, method='ward')
    plt.figure(figsize=(10, 7))
    dendrogram(Z)
    plt.title('Dendrogram')
    plt.xlabel('Data Points')
    plt.ylabel('Distance')
    plt.savefig(output_file)
    plt.close()
    logger.info(f"Dendrogram plot saved to {output_file}")

def compute_dendrogram(embeddings_file: Path, dendrogram_file: Path):
    files, vectors = load_embeddings(file=embeddings_file)
    plot_dendrogram(vectors, dendrogram_file)

def set_paths_and_run(embeddings_file: str, dendrogram_file: str):
    compute_dendrogram(
        embeddings_file=Path(embeddings_file),
        dendrogram_file=Path(dendrogram_file)
    )

if __name__ == "__main__":
    # Example programmatic usage
    embeddings_file = "/Users/vishanthsuresh/Downloads/Data Science/Dissertation/
 ↪PhIDDLI-main/data/reduced_UMAP.pkl"
    dendrogram_file = "/Users/vishanthsuresh/Downloads/Data Science/Dissertation/
 ↪PhIDDLI-main/data/dendrogram.png"

    set_paths_and_run(embeddings_file, dendrogram_file)

    # Uncomment the following lines if you want to use command-line arguments␣
 ↪instead
    # parser = ArgumentParser()
    # parser.add_argument('embeddings_file', type=Path)
    # parser.add_argument('--dendrogram-file', type=Path, required=True)
    # args = parser.parse_args()
    # compute_dendrogram(
    #     embeddings_file=args.embeddings_file,
    #     dendrogram_file=args.dendrogram_file)
```

```
2024-07-13 16:04:54.269 | INFO     |
__main__:plot_dendrogram:26 - Dendrogram plot
saved to /Users/vishanthsuresh/Downloads/Data Science/Dissertation/PhIDDLI-
main/data/dendrogram.png
```

```
[8]: import pickle
from scipy.cluster.hierarchy import linkage, dendrogram
import matplotlib.pyplot as plt
import numpy as np

# Define the path to the embeddings file
embeddings_file_path = '/Users/vishanthsuresh/Downloads/Data Science/
 ↪Dissertation/PhIDDLI-main/data/reduced_UMAP.pkl'

# Load the embeddings from the pickle file
with open(embeddings_file_path, 'rb') as file:
    embeddings = pickle.load(file)

# Extract the vectors from the loaded data
files, vectors = zip(*list(embeddings.items()))
vectors = np.array(vectors)  # Ensure vectors are a NumPy array

# Compute the linkage matrix
Z = linkage(vectors, method='ward')

# Plot the dendrogram without data point markers
plt.figure(figsize=(6, 4))
plt.title("Dendrogram UMAP-Hierarchical")
dendrogram(Z, leaf_rotation=90, leaf_font_size=10, no_labels=True)
plt.xlabel('Cluster Size')
plt.ylabel('Distance')
plt.show()
```



2.0.1 Choosing the optimal number of Clusters

From Dendogram the number of optimal clusters is chosen as 6 as the thershold line at the lowest point of the longest vertical line passes through six lines.

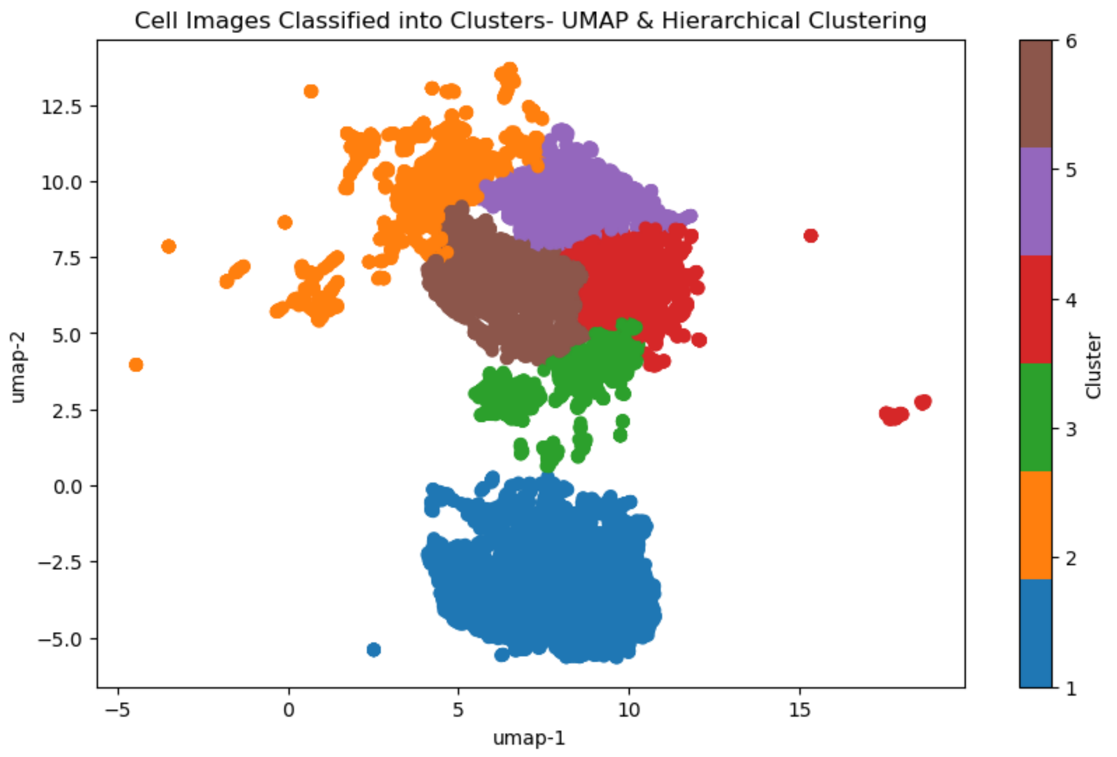
[39]:
```
import pickle
from argparse import ArgumentParser
from pathlib import Path
from typing import List, Optional, Tuple

import numpy as np
from loguru import logger
from tqdm import tqdm
from scipy.cluster.hierarchy import linkage, fcluster
import hdbscan

def load_embeddings(file: Path) -> Tuple[List[Path], np.ndarray]:
    with file.open('rb') as f:
        embeddings = pickle.load(f)
    files, vectors = zip(*list(embeddings.items()))
    vectors = np.array(vectors)  # Ensure vectors are a NumPy array
    return files, vectors

def make_clusters(vectors: np.ndarray, n_clusters: int) -> np.ndarray:
    Z = linkage(vectors, method='ward')
    logger.info(f"Performing hierarchical clustering with {n_clusters} clusters")
    return fcluster(Z, n_clusters, criterion='maxclust')

def find_optimal_number_of_clusters(data: np.ndarray) -> int:
    #logger.info("Finding optimal value for k using HDBSCAN ...")
    #clusterer = hdbscan.HDBSCAN(min_cluster_size=100)
    #clusterer.fit(data)

    #labels = clusterer.labels_
    #num_clusters = len(set(labels)) - (1 if -1 in labels else 0)
    #logger.info(f"Optimal number of clusters determined by HDBSCAN:␣
 ↪{num_clusters}")
    #return num_clusters
    return 6

def compute_clusters(embeddings_file: Path, output_file: Path, clusters:␣
 ↪Optional[int] = None):
    files, vectors = load_embeddings(file=embeddings_file)
    if clusters is None:
        clusters = find_optimal_number_of_clusters(data=vectors)
    cluster_labels = make_clusters(vectors=vectors, n_clusters=clusters)
    cluster_mapping = dict(zip(files, cluster_labels))
    with output_file.open('wb') as f:
        pickle.dump(cluster_mapping, f)

def set_paths_and_run(embeddings_file: str, output_file: str, clusters:␣
 ↪Optional[int] = None):
    compute_clusters(
        embeddings_file=Path(embeddings_file),
        output_file=Path(output_file),
        clusters=clusters
    )

if __name__ == "__main__":
    # Example programmatic usage
    embeddings_file = "/Users/vishanthsuresh/Downloads/Data Science/Dissertation/
 ↪PhIDDLI-main/data/reduced_UMAP.pkl"
    output_file = "/Users/vishanthsuresh/Downloads/Data Science/Dissertation/
 ↪PhIDDLI-main/data/rHierarichal_clusters_UMAP.pkl"
    clusters = None  # Or specify an integer for a fixed number of clusters

    set_paths_and_run(embeddings_file, output_file, clusters)

    # Uncomment the following lines if you want to use command-line arguments␣
 ↪instead
    # parser = ArgumentParser()
    # parser.add_argument('embeddings_file', type=Path)
    # parser.add_argument('--clusters', type=lambda x: int(x) if x else None,␣
↪default=None, required=False)
    # parser.add_argument('--output-file', type=Path, required=True)
    # args = parser.parse_args()
    # compute_clusters(
    #     embeddings_file=args.embeddings_file,
    #     output_file=args.output_file,
    #     clusters=args.clusters)
```

```
2024-08-07 23:17:57.305 | INFO     |
__main__:make_clusters:21 - Performing

hierarchical clustering with 6 clusters
```

[52]:
```
import pickle
from argparse import ArgumentParser
from collections import defaultdict
from pathlib import Path
from typing import List, Tuple

import pandas as pd

def load_file_contents(file: Path) -> Tuple[List[Path], List]:
    return pickle.loads(file.read_bytes())

def export_csv(clusters_file: Path, points_file: Path, output_file: Path):
    clusters_data = load_file_contents(clusters_file)
    points_data = load_file_contents(points_file)

    data = defaultdict(dict)
    for key, cluster in clusters_data.items():
        data[key]["cluster"] = cluster
        point = points_data[key]
        data[key]["point_x"], data[key]["point_y"] = point

    df: pd.DataFrame = pd.DataFrame.from_dict(data, orient="index") \
        .rename_axis('cell_location') \
        .reset_index() \
        .assign(cell_location=lambda df: df.cell_location.apply(Path)) \
        .assign(parent_image=lambda df: df.cell_location.apply(lambda path: path.
↪parent.name))

    df.to_csv(output_file, index=False)

def set_paths_and_run(clusters_file: str, points_file: str, output_file: str):
    export_csv(
        clusters_file=Path(clusters_file),
        points_file=Path(points_file),
        output_file=Path(output_file)
    )

if __name__ == "__main__":
    # Example programmatic usage
    clusters_file = "/Users/vishanthsuresh/Downloads/Data Science/Dissertation/
 ↪PhIDDLI-main/data/rHierarichal_clusters_UMAP.pkl"
    points_file = "/Users/vishanthsuresh/Downloads/Data Science/Dissertation/
 ↪PhIDDLI-main/data/reduced_UMAP.pkl"
    output_file = "/Users/vishanthsuresh/Downloads/Data Science/Dissertation/
 ↪PhIDDLI-main/data/output_hierarchical_umap.csv"

    set_paths_and_run(clusters_file, points_file, output_file)

    # Uncomment the following lines if you want to use command-line arguments␣
 ↪instead
    # parser = ArgumentParser()
    # parser.add_argument('--clusters', type=Path, required=True)
    # parser.add_argument('--points', type=Path, required=True)
    # parser.add_argument('--output-file', type=Path, required=True)
    # args = parser.parse_args()
    # export_csv(
    #     clusters_file=args.clusters,
    #     points_file=args.points,
    #     output_file=args.output_file)
```

```
[7]: import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.colors as mcolors


data= pd.read_csv("/Users/vishanthsuresh/Downloads/Data Science/Dissertation/
 ↪PhIDDLI-main/data/output_hierarchical_umap.csv")
# Define a set of more contrasting and visually distinct colors
contrasting_colors = [
    '#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd',
    '#8c564b'
]

# Create a colormap from the list of contrasting colors
custom_cmap_contrasting = mcolors.ListedColormap(contrasting_colors[:15])

# Plotting point_x and point_y with clusters in different colors using the␣
 ↪custom colormap
plt.figure(figsize=(10, 6))
```

```
# Using a scatter plot to visualize the clusters
scatter = plt.scatter(data['point_x'], data['point_y'], c=data['cluster'],␣
↪cmap=custom_cmap_contrasting)

# Adding a color bar to indicate the cluster colors
cbar = plt.colorbar(scatter, label='Cluster', ticks=range(15))
cbar.set_ticklabels(range(15))

# Adding titles and labels
plt.title('Cell Images Classified into Clusters- UMAP & Hierarchical Clustering')
plt.xlabel('umap-1')
plt.ylabel('umap-2')

# Display the plot
plt.show()
```



[55]:
```
import pandas as pd
from sklearn.metrics import silhouette_score, calinski_harabasz_score,␣
↪davies_bouldin_score

# Load the CSV file
```

```
file_path = '/Users/vishanthsuresh/Downloads/Data Science/Dissertation/
 ↪PhIDDLI-main/data/output_hierarchical_umap.csv'
data = pd.read_csv(file_path)

# Extracting the relevant data for clustering validation
vectors = data[['point_x', 'point_y']].values
labels = data['cluster'].values

# Calculate the validation metrics
silhouette_avg = silhouette_score(vectors, labels)
calinski_harabasz_avg = calinski_harabasz_score(vectors, labels)
davies_bouldin_avg = davies_bouldin_score(vectors, labels)

print(f"Silhouette Score: {silhouette_avg}")
print(f"Calinski-Harabasz Index: {calinski_harabasz_avg}")
print(f"Davies-Bouldin Index: {davies_bouldin_avg}")
```

```
Silhouette Score: 0.41658099104234314
Calinski-Harabasz Index: 25387.171079316387
Davies-Bouldin Index: 0.8338531148866153
```

[5]:
```
import os
import shutil
import pandas as pd

# Load the data
file_path = '/Users/vishanthsuresh/Downloads/Data Science/Dissertation/
 ↪PhIDDLI-main/data/output_hierarchical_umap.csv'
data = pd.read_csv(file_path)

# Base directory where images are stored
base_image_dir = '/Users/vishanthsuresh/Downloads/Data Science/Dissertation/
 ↪PhIDDLI-main/data/extracted_cells'

# Base directory to store segregated clusters
output_dir = '/Users/vishanthsuresh/Downloads/Data Science/Dissertation/
 ↪PhIDDLI-main/data/clusters_umap_hierarchical'

# Function to get all image paths from subdirectories
def get_all_image_paths(base_dir):
    image_paths = []
    for root, _, files in os.walk(base_dir):
        for file in files:
            if file.endswith(('.jpg', '.jpeg', '.png', '.tif', '.tiff')):  # Add␣
 ↪other image extensions if needed
                image_paths.append(os.path.join(root, file))
    return image_paths
```

```
# Get all image paths from the base image directory
all_image_paths = get_all_image_paths(base_image_dir)

# Create a dictionary to map relative cell locations to their paths
image_path_dict = {os.path.relpath(path, base_image_dir): path for path in␣
↪all_image_paths}

# Create directories for each cluster
unique_clusters = data['cluster'].unique()
for cluster in unique_clusters:
    cluster_dir = os.path.join(output_dir, f'cluster_{cluster}')
    if not os.path.exists(cluster_dir):
        os.makedirs(cluster_dir)

# Move files to corresponding cluster directories
for index, row in data.iterrows():
    cluster = row['cluster']
    relative_path = os.path.relpath(row['cell_location'], base_image_dir)
    source_file_path = image_path_dict.get(relative_path)

    if source_file_path and os.path.exists(source_file_path):
        # Create a unique destination file path
        unique_filename = f"{os.path.splitext(relative_path.replace('/',␣
↪'_'))[0]}_{index}{os.path.splitext(relative_path)[1]}"
        dest_file_path = os.path.join(output_dir, f'cluster_{cluster}',␣
↪unique_filename)

        # Ensure the directory exists
        dest_dir = os.path.dirname(dest_file_path)
        if not os.path.exists(dest_dir):
            os.makedirs(dest_dir)

        shutil.copy(source_file_path, dest_file_path)
    else:
        print(f"File not found for: {relative_path}")

print("Images segregated into cluster folders successfully.")
```

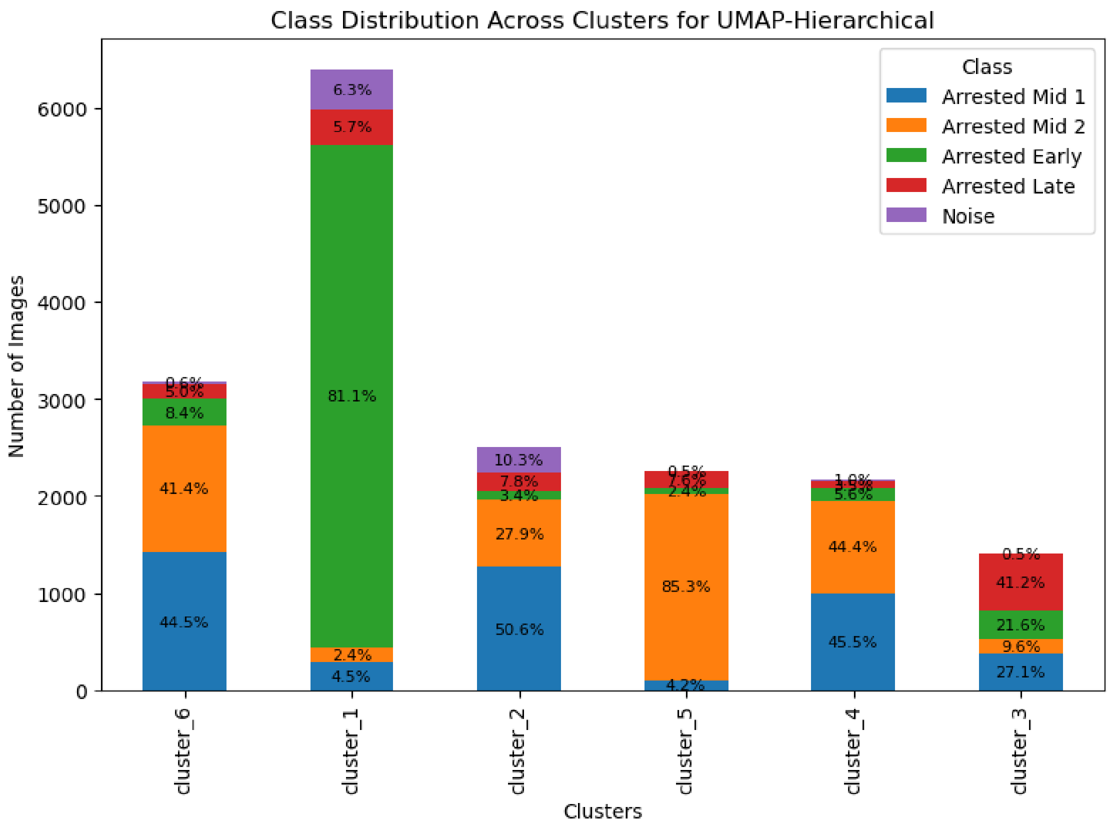
```
Images segregated into cluster folders successfully.
```

[6]:
```
import os
import torch
import torch.nn as nn
from torchvision import transforms
from PIL import Image
import pandas as pd
```

```python
import matplotlib.pyplot as plt

# Define CNN Model with three hidden layers (same as before)
class CustomCNN(nn.Module):
    def __init__(self):
        super(CustomCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.maxpool1 = nn.MaxPool2d(kernel_size=2)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.conv4 = nn.Conv2d(128, 128, kernel_size=3, padding=1)
        self.maxpool2 = nn.MaxPool2d(kernel_size=2)
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(128 * 8 * 8, 512)
        self.fc2 = nn.Linear(512, 256)  # Third hidden layer
        self.fc3 = nn.Linear(256, 128)  # Fourth hidden layer
        self.fc4 = nn.Linear(128, 30)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.maxpool1(self.relu(self.conv1(x)))
        x = self.maxpool1(self.relu(self.conv2(x)))
        x = self.relu(self.conv3(x))
        x = self.maxpool2(self.relu(self.conv4(x)))
        x = self.flatten(x)
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))  # Third hidden layer
        x = self.relu(self.fc3(x))  # Fourth hidden layer
        x = self.fc4(x)
        return x

# Load the checkpoint
checkpoint = torch.load('/Users/vishanthsuresh/Downloads/Data Science/
 ↪Dissertation/CNN Model/final_model_checkpoint.pth')
model = CustomCNN()
model.load_state_dict(checkpoint['model_state_dict'])
model.eval()

# Define the transformation for the images
transform = transforms.Compose([
    transforms.Resize((64, 64)),
    transforms.ToTensor(),
])

# Define a dictionary to map class indices to class names
class_names = {
    0: 'Arrested Mid 1',
    1: 'Arrested Mid 2',
    2: 'Arrested Early',
    3: 'Arrested Late',
    4: 'Noise'
    # Add other class mappings as necessary
}

# Function to predict image
def predict_image(image_path, model, transform):
    image = Image.open(image_path)
    image = transform(image)
    image = image.unsqueeze(0)  # Add batch dimension
    with torch.no_grad():
        output = model(image)
    _, predicted = torch.max(output, 1)
    return class_names[predicted.item()]

# Directory containing all the clusters
base_folder = '/Users/vishanthsuresh/Downloads/Data Science/Dissertation/
 ↪PhIDDLI-main/data/clusters_umap_hierarchical'
# Initialize an empty DataFrame with columns for each class
all_class_counts = pd.DataFrame(columns=class_names.values())
cluster_names = []

# Iterate over all subfolders and predict images
for folder_name in os.listdir(base_folder):
    folder_path = os.path.join(base_folder, folder_name)
    if os.path.isdir(folder_path):  # Ensure it's a directory
        results = []
        for image_name in os.listdir(folder_path):
            if image_name.endswith(('.png', '.jpg', '.jpeg')):  # Add other␣
 ↪image extensions if needed
                image_path = os.path.join(folder_path, image_name)
                predicted_class = predict_image(image_path, model, transform)
                results.append({'Image': image_name, 'Predicted_Class':␣
 ↪predicted_class})

        # Save results to CSV
        results_df = pd.DataFrame(results)
        results_csv_path = os.path.join(folder_path, f'{folder_name}_predictions.
 ↪csv')
        results_df.to_csv(results_csv_path, index=False)
        print(f"Results saved to {results_csv_path}")

        # Store class counts for the current cluster
        class_counts = results_df['Predicted_Class'].value_counts()
        # Add missing classes with 0 count
        class_counts = class_counts.reindex(all_class_counts.columns,␣
↪fill_value=0)
        all_class_counts = pd.concat([all_class_counts, class_counts.to_frame().
↪T], ignore_index=True)
        cluster_names.append(folder_name)

# Set index to cluster names
all_class_counts.index = cluster_names

# Plot stacked bar chart with percentage annotations
fig, ax = plt.subplots(figsize=(8, 6))
bars = all_class_counts.plot(kind='bar', stacked=True, ax=ax)

# Calculate percentages and annotate
for i, cluster in enumerate(all_class_counts.index):
    total = all_class_counts.loc[cluster].sum()
    cumulative_sum = 0
    for j, class_name in enumerate(all_class_counts.columns):
        class_count = all_class_counts.loc[cluster, class_name]
        cumulative_sum += class_count
        if class_count > 0:
            percentage = (class_count / total) * 100
            ax.text(i, cumulative_sum - class_count / 2, f'{percentage:.1f}%',␣
↪ha='center', va='center', fontsize=8)

plt.title('Class Distribution Across Clusters for UMAP-Hierarchical')
plt.xlabel('Clusters')
plt.ylabel('Number of Images')
plt.legend(title='Class')
plt.tight_layout()
plt.show()
```

```
Results saved to /Users/vishanthsuresh/Downloads/Data
Science/Dissertation/PhIDDLI-
main/data/clusters_umap_hierarchical/cluster_6/cluster_6_predictions.csv
Results saved to /Users/vishanthsuresh/Downloads/Data
Science/Dissertation/PhIDDLI-
main/data/clusters_umap_hierarchical/cluster_1/cluster_1_predictions.csv
Results saved to /Users/vishanthsuresh/Downloads/Data
Science/Dissertation/PhIDDLI-
main/data/clusters_umap_hierarchical/cluster_2/cluster_2_predictions.csv
Results saved to /Users/vishanthsuresh/Downloads/Data
Science/Dissertation/PhIDDLI-
main/data/clusters_umap_hierarchical/cluster_5/cluster_5_predictions.csv
Results saved to /Users/vishanthsuresh/Downloads/Data
Science/Dissertation/PhIDDLI-
main/data/clusters_umap_hierarchical/cluster_4/cluster_4_predictions.csv
```

```
Results saved to /Users/vishanthsuresh/Downloads/Data
Science/Dissertation/PhIDDLI-
main/data/clusters_umap_hierarchical/cluster_3/cluster_3_predictions.csv
```



The images for all the clusters are segregated and stored on the Onedrive. Link has been provided in the email

Semi Supervised Fuzzy Clustering

September 2, 2024

0.0.1 Semi Supervised Fuzzy Clustering

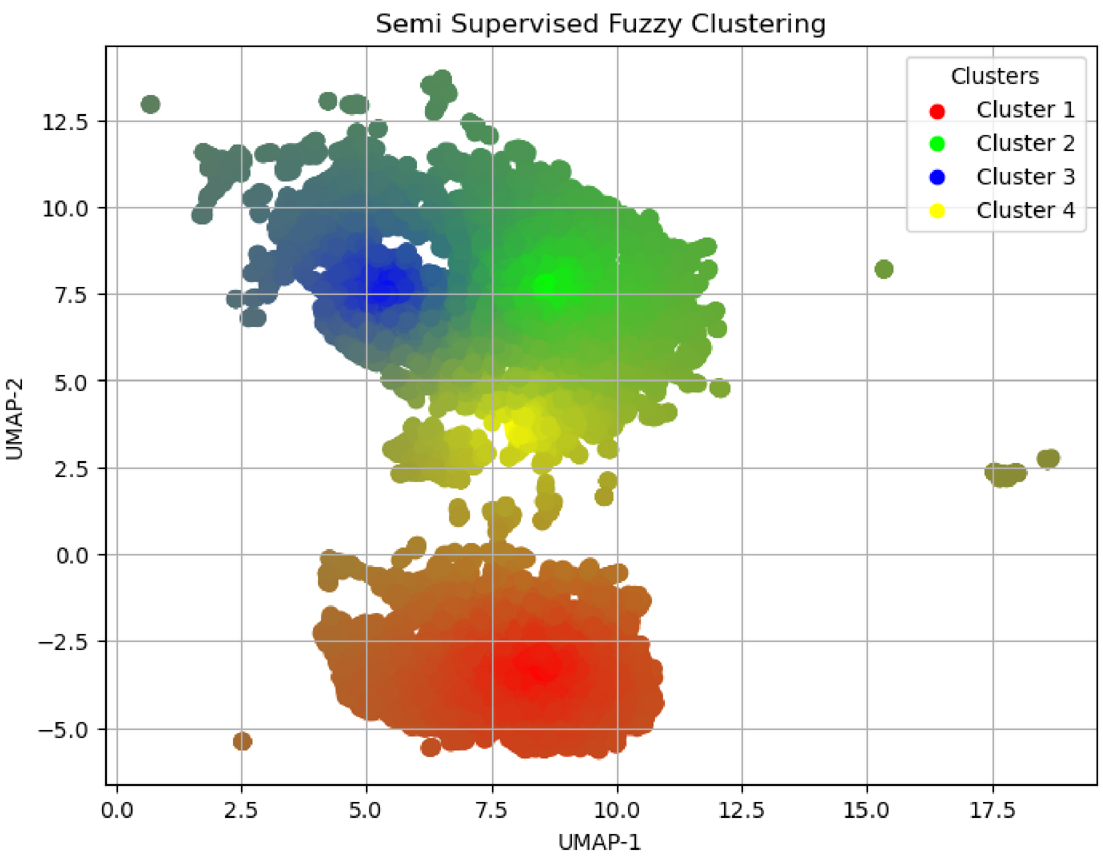
[31]:
```
import pandas as pd
import numpy as np
from sklearn.preprocessing import normalize
from skfuzzy import cmeans

# Load the UMAP-reduced embeddings
umap_embeddings = pd.read_csv('/Users/vishanthsuresh/Downloads/Data Science/
↪Dissertation/PhIDDLI-main/data/UMAP_fuzzy_data_points.csv')
data_points = umap_embeddings[['point_x', 'point_y']].values

# Load the labeled data
labeled_data_df = pd.read_csv('/Users/vishanthsuresh/Downloads/Data Science/
↪Dissertation/PhIDDLI-main/data/partial_label_V2.0.csv')  # Update with actual␣
↪path to the labeled data CSV
labeled_data_points = labeled_data_df[['point_x', 'point_y']].values
labels = labeled_data_df['class'].values

# Combine data points and labeled data for initial clustering
combined_data = np.vstack([data_points, labeled_data_points])

# Define Fuzzy C-Means parameters
n_clusters = len(np.unique(labels))  # Number of clusters (same as number of␣
↪unique classes)
fuzziness = 3.0  # Fuzzification parameter
max_iter = 100
error = 0.005
alpha = 0.5  # Weight for partial supervision

# Perform initial Fuzzy C-Means clustering
cntr, u, _, _, _, _, _ = cmeans(combined_data.T, n_clusters, fuzziness,␣
↪error=error, maxiter=max_iter)

# Update membership matrix u with partial supervision
labeled_count = len(labeled_data_points)
full_count = len(combined_data)
```

```
# Create a mapping from class names to cluster indices
class_names = np.unique(labels)
class_to_cluster_index = {class_name: idx for idx, class_name in␣
 ↪enumerate(class_names)}

for j in range(labeled_count):
    class_name = labels[j]
    cluster_index = class_to_cluster_index[class_name]
    u[cluster_index, -labeled_count + j] = 1

u = normalize(u, norm='l1', axis=0)

# Extract the fuzzy membership values for the original data points
membership_values = u[:, :-labeled_count].T

# Add membership values for each cluster to the DataFrame
for idx in range(n_clusters):
    umap_embeddings[f'membership_{idx + 1}'] = membership_values[:, idx]

# Add cluster labels to the DataFrame (label clusters from 1 to 5)
umap_embeddings['cluster'] = np.argmax(membership_values, axis=1) + 1

# Save the results to the same CSV file
umap_embeddings.to_csv('/Users/vishanthsuresh/Downloads/Data Science/
 ↪Dissertation/PhIDDLI-main/data/UMAP_fuzzy_data_points.csv', index=False)

print('Clustering completed and results saved.')
```

```
Clustering completed and results saved.
```

```
[48]: import matplotlib.pyplot as plt
import numpy as np
import mplcursors

data= pd.read_csv("/Users/vishanthsuresh/Downloads/Data Science/Dissertation/
 ↪PhIDDLI-main/data/UMAP_fuzzy_data_points.csv")
# Assuming 'data' is already loaded as shown in your code
# Extract relevant data
x = data['point_x']
y = data['point_y']
memberships = data[['membership_1', 'membership_2', 'membership_3',␣
 ↪'membership_4']]

# Normalize membership values to get a color for each point
colors = memberships.values.dot(np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1], [1,␣
 ↪1, 0]]))
colors = colors / np.max(colors, axis=0)  # Normalize colors to [0, 1]
```

```python
# Create the scatter plot
plt.figure(figsize=(8, 6))
scatter = plt.scatter(x, y, c=colors, s=50)

# Manually add legend for each cluster
for i, color in enumerate([[1, 0, 0], [0, 1, 0], [0, 0, 1], [1, 1, 0]]):
    plt.scatter([], [], c=[color], label=f'Cluster {i+1}')

plt.title('Semi Supervised Fuzzy Clustering')
plt.xlabel('UMAP-1')
plt.ylabel('UMAP-2')
plt.legend(title="Clusters")
plt.grid(True)

# Add interactive cursor
cursor = mplcursors.cursor(scatter, hover=True)

# Display the coordinates when a point is clicked
@cursor.connect("add")
def on_add(sel):
    sel.annotation.set_text(f"({sel.target[0]:.2f}, {sel.target[1]:.2f})")

plt.show()
```


Semi Supervised Fuzzy Clustering
Clusters
Cluster 1
Cluster 2
Cluster 3
Cluster 4
UMAP-1
UMAP-2
12.5
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5