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

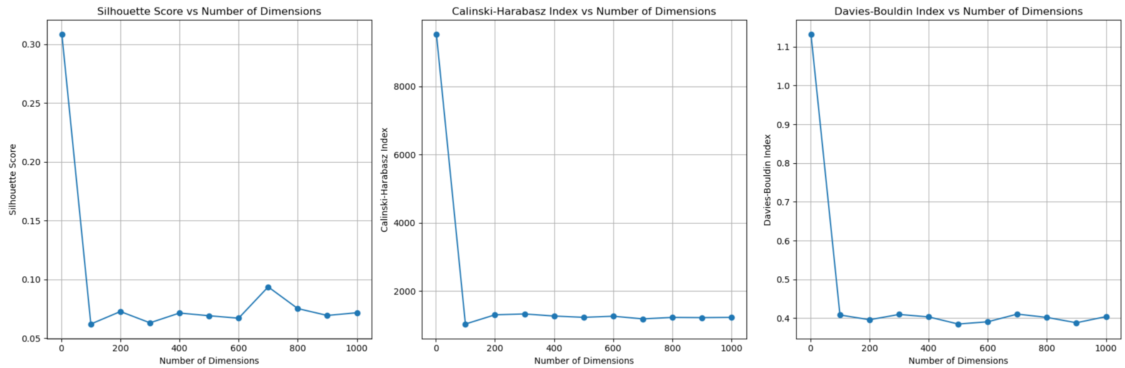
[5]:
```
import numpy as np
import pandas as pd
from sklearn.neighbors import NearestNeighbors
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score, calinski_harabasz_score,␣
↪davies_bouldin_score
import matplotlib.pyplot as plt
import random

# 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
```

```python
    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:
```

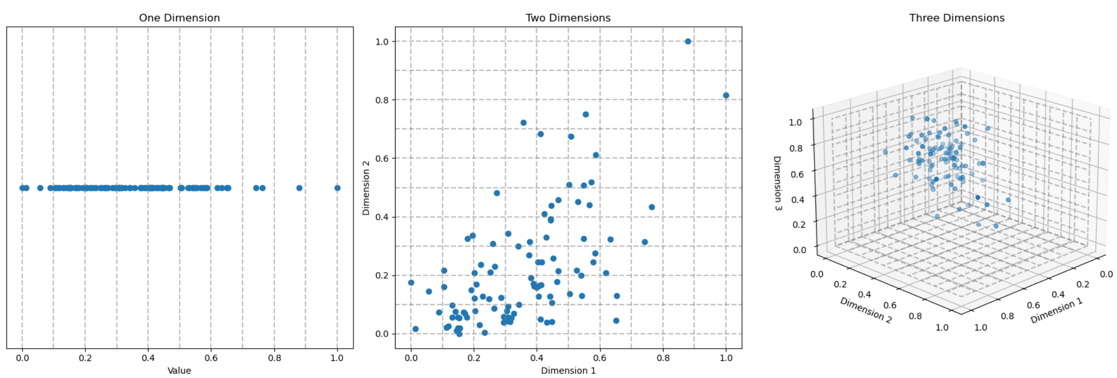
```
1220.3707033531489, Davies-Bouldin Index: 0.3881637214843754
Dimensions: 1000, Silhouette Score: 0.07156348938036947, Calinski-Harabasz
Index: 1227.3582978178738, Davies-Bouldin Index: 0.40395099656107053
```



[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}")
```



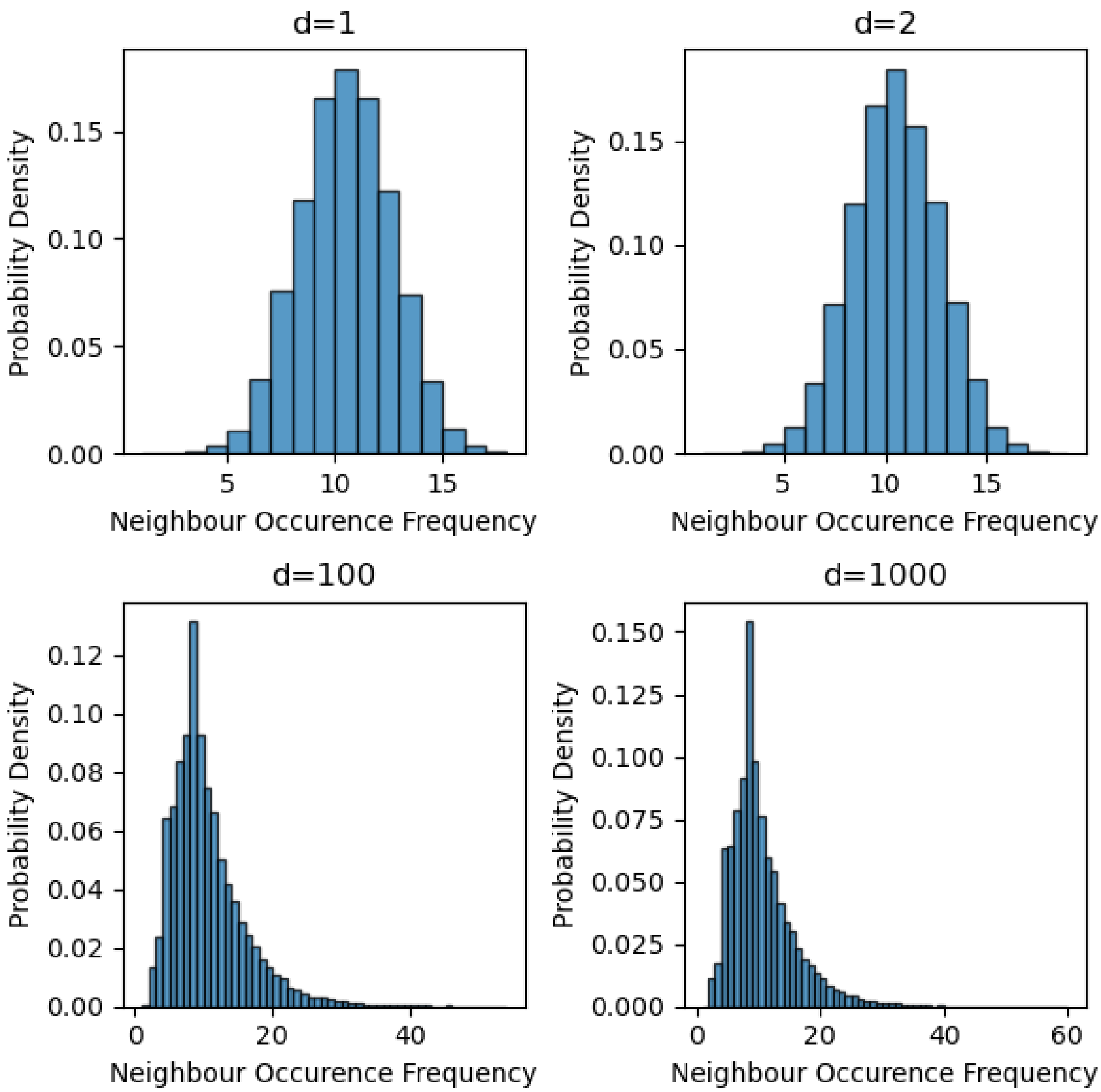

```
One Dimension: 1 empty cells out of 10
Two Dimensions: 65 empty cells out of 100
Three Dimensions: 928 empty cells out of 1000
```

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')
```

```python
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)
```

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



[64]:
```python
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