assignment-5-copy1

December 5, 2023

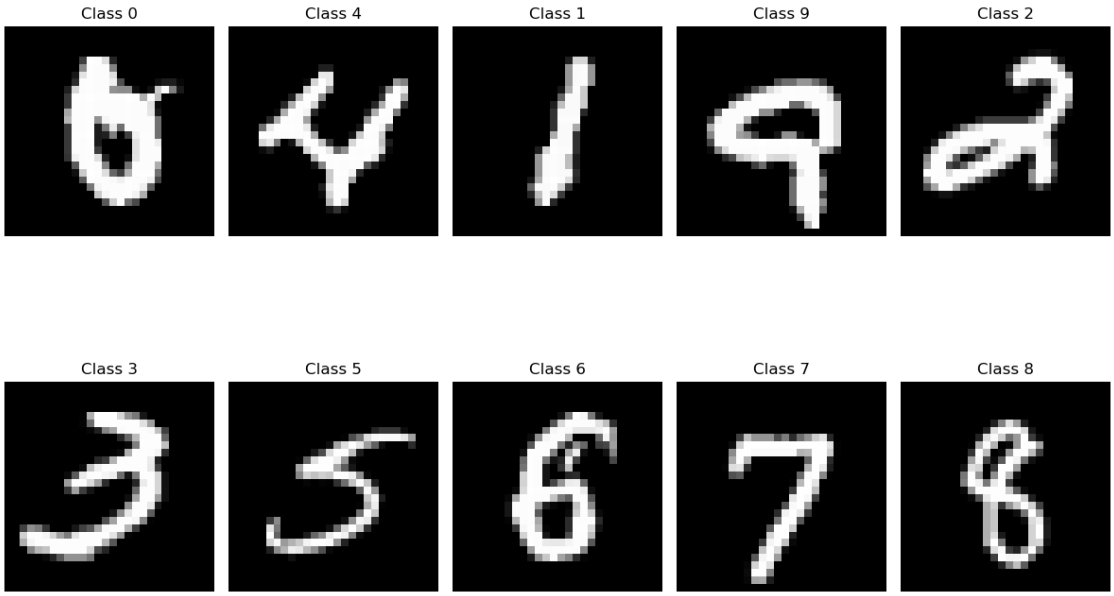
1 Late Days Used: 1 | CS 57300: Assignment 5

Name: Ishaan Roychowdhury
Date: November 17, 2023

1.1 1) Exploration (5 pts)

1.1) Randomly pick one digit from each class in digits-raw.csv and visualize its image as a 28×28 grayscale matrix.

[1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
data = pd.read_csv('digits-raw.csv')
unique_classes = data.iloc[:, 1].unique()
np.random.seed(0)
num_rows = len(unique_classes) // 5 + 1
fig, axs = plt.subplots(num_rows, 5, figsize=(12, 12))
for i, class_label in enumerate(unique_classes):
    chosen_one = data[data.iloc[:, 1] == class_label].sample(n=1)
    matrix_chosen_for_Q1 = chosen_one.iloc[:, 2:].values.reshape(28, 28)
    row, col = i // 5, i % 5
    axs[row, col].imshow(matrix_chosen_for_Q1, cmap='gray')
    axs[row, col].set_title(f'Class {class_label}')
    axs[row, col].axis('off')
for i in range(len(unique_classes), num_rows * 5):
    row, col = i // 5, i % 5
    fig.delaxes(axs[row, col])
plt.tight_layout()
plt.show()
```



1.2) Visualize 1000 randomly selected examples in 2D from the file digits-embedding.csv, coloring the points to show their corresponding class labels. Select the examples randomly using the command np.random.randint(0, N, size=1000), where N is the total number of examples.

[2]:
```
from matplotlib.colors import ListedColormap
df = pd.read_csv('digits-embedding.csv')
df.head()
```

[2]:

	0	5	-13.9383184	-20.94943593
0	1	0	-26.283715	16.836709
1	2	4	36.899612	-19.903789
2	3	1	12.913048	-16.954055
3	4	9	22.947148	-2.747205
4	5	2	-9.380422	35.386284

[3]:
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
N = df.shape[0]
given_sample = np.random.randint(0, N, size=1000)
chosen_samples_for_q1 = df.iloc[given_sample]
x = chosen_samples_for_q1.iloc[:, 2]
y = chosen_samples_for_q1.iloc[:, 3]
label_of_Class = chosen_samples_for_q1.iloc[:, 1].astype(int)
num_classes = len(np.unique(label_of_Class))
```

```
colors = ['#' + ''.join(np.random.choice(list('0123456789ABCDEF'), size=6)) for␣
 ↪_ in range(num_classes)]
cmap = ListedColormap(colors)
plt.figure(figsize=(10, 8))
finalPlot = plt.scatter(x, y, c=label_of_Class, cmap=cmap, alpha=0.8)
cb = plt.colorbar(finalPlot, ticks=np.arange(num_classes))
cb.set_label('Class Labels')
cb.ax.set_yticklabels(np.unique(label_of_Class))
plt.title('2D Visualization of 1000 Randomly Selected Digits')
plt.xlabel('Dimension 1 of Embedding')
plt.ylabel('Dimension 2 of Embedding')
plt.show()
```

1.2 2) K-means Clustering (30 pts)

2.1) Code (10 pts) Your implementation should include a function kmeans(·), which takes dataFilename, K and max iter as input, and outputs the three evaluation metrics shared above. Note that dataFilename corresponds to a subset of the embedding data (in the same format as digits-embedding.csv), K corresponds to the number of clusters to use, and max iter corresponds to number of iterations before terminating the algorithm

[4]:
```
data = pd.read_csv('digits-embedding.csv', header=None)
data.columns = ['image_id', 'class_label', 'embedding_feature1',␣
 ↪'embedding_feature2']
data.head()
```

[4]:

	image_id	class_label	embedding_feature1	embedding_feature2
0	0	5	-13.938318	-20.949436
1	1	0	-26.283715	16.836709
2	2	4	36.899612	-19.903789
3	3	1	12.913048	-16.954055
4	4	9	22.947148	-2.747205

[5]:
```
def kmeans(data, K, max_iter):
    np.random.seed(0)
    X = data.iloc[:, 2:].values
    correct_output_label = data.iloc[:, 1].values
    centroids_for_part2 = X[np.random.choice(X.shape[0], K, replace=False)]
    WC_SSD = 0
    for _ in range(max_iter):
        distances = np.sqrt(((X - centroids_for_part2[:, np.newaxis]) ** 2).
 ↪sum(axis=2)).T
        argMinDistances = np.argmin(distances, axis=1)
        new_mapping = []
        WC_SSD = 0
        for k in range(K):
            m = argMinDistances == k
            if np.any(m):
                new_n = X[m]
                new_c = new_n.mean(axis=0)
                new_mapping.append(new_c)
                WC_SSD += np.sum((new_n - centroids_for_part2[k])**2)
            else:
                total_euc_dist = np.sum(np.sqrt(((X - centroids_for_part2[:, np.
 ↪newaxis]) ** 2).sum(axis=2)).T, axis=1)
                far_pt = np.argmax(total_euc_dist)
                new_mapping.append(X[far_pt])
        new_mapping = np.array(new_mapping)
        if np.allclose(centroids_for_part2, new_mapping, atol=1e-4):
            break
        centroids_for_part2 = new_mapping
```

```
    sc_val_arr = []
    for i in range(X.shape[0]):
        c_index = argMinDistances[i]
        internal_dist = X[argMinDistances == c_index]
        inter_c_dist = np.sqrt(np.sum((internal_dist - X[i])**2, axis=1))
        a_i = np.mean(inter_c_dist)
        diff_C = [c for c in range(centroids_for_part2.shape[0]) if c !=␣
↪c_index]
        dist_to_nearest_cluster = []
        for c in diff_C:
            neighboring_cluster_points = X[argMinDistances == c]
            distances_to_neighboring_cluster = np.sqrt(np.
↪sum((neighboring_cluster_points - X[i])**2, axis=1))
            dist_to_nearest_cluster.append(np.
↪mean(distances_to_neighboring_cluster))
        b_i = np.mean(dist_to_nearest_cluster) if dist_to_nearest_cluster else 0
        sc_i = (b_i - a_i) / max(a_i, b_i) if max(a_i, b_i) > 0 else 0
        sc_val_arr.append(sc_i)
    SC = np.mean(sc_val_arr)
    correct_label_i = np.unique(correct_output_label, return_inverse=True)[1]
    pred_i = np.unique(argMinDistances, return_inverse=True)[1]
    matrix_for_nmi = np.histogram2d(correct_label_i, pred_i,
                                    bins=[np.arange(np.
↪max(correct_label_i)+2),
                                          np.arange(np.max(pred_i)+2)])[0]
    sum_t = matrix_for_nmi.sum(axis=1)
    sum_pred = matrix_for_nmi.sum(axis=0)
    total = matrix_for_nmi.sum()
    entropy_t = -np.sum((sum_t / total) * np.log2(sum_t / total + 1e-15))
    entropy_pred = -np.sum((sum_pred / total) * np.log2(sum_pred / total +␣
↪1e-15))
    with np.errstate(divide='ignore', invalid='ignore'):
        joint_prob = matrix_for_nmi / total
        prod_prob = sum_t[:, None] * sum_pred[None, :] / total**2
        mutual_information = np.nansum(joint_prob * np.log2(joint_prob /␣
↪prod_prob + 1e-15))
    NMI = mutual_information / (entropy_t + entropy_pred)
    print("WC-SSD:", WC_SSD)
    print("SC:", SC)
    print("NMI:", NMI)
```

[6]:
```
data = 'digits-embedding.csv'
dataFilename = pd.read_csv(data)
kmeans(dataFilename, 10, 50)
```

```
WC-SSD: 1466235.0349835493
SC: 0.711734367320828
```

```
NMI: 0.3666514450504181
```

2.2) Analysis (20 pts)

[7]:
```
def kmeans(data, K, max_iter):
    np.random.seed(0)
    X = data.iloc[:, 2:].values
    correct_output_label = data.iloc[:, 1].values
    centroids_for_part2 = X[np.random.choice(X.shape[0], K, replace=False)]
    WC_SSD = 0
    for _ in range(max_iter):
        distances = np.sqrt(((X - centroids_for_part2[:, np.newaxis]) ** 2).
↪sum(axis=2)).T
        argMinDistances = np.argmin(distances, axis=1)
        new_mapping = []
        WC_SSD = 0
        for k in range(K):
            m = argMinDistances == k
            if np.any(m):
                new_n = X[m]
                new_c = new_n.mean(axis=0)
                new_mapping.append(new_c)
                WC_SSD += np.sum((new_n - centroids_for_part2[k])**2)
            else:
                total_euc_dist = np.sum(np.sqrt(((X - centroids_for_part2[:, np.
↪newaxis]) ** 2).sum(axis=2)).T, axis=1)
                far_pt = np.argmax(total_euc_dist)
                new_mapping.append(X[far_pt])
        new_mapping = np.array(new_mapping)
        if np.allclose(centroids_for_part2, new_mapping, atol=1e-4):
            break
        centroids_for_part2 = new_mapping
    sc_val_arr = []
    for i in range(X.shape[0]):
        c_index = argMinDistances[i]
        internal_dist = X[argMinDistances == c_index]
        inter_c_dist = np.sqrt(np.sum((internal_dist - X[i])**2, axis=1))
        a_i = np.mean(inter_c_dist)
        diff_C = [c for c in range(centroids_for_part2.shape[0]) if c !=␣
↪c_index]
        dist_to_nearest_cluster = []
        for c in diff_C:
            neighboring_cluster_points = X[argMinDistances == c]
            distances_to_neighboring_cluster = np.sqrt(np.
↪sum((neighboring_cluster_points - X[i])**2, axis=1))
```

```
            dist_to_nearest_cluster.append(np.
↪mean(distances_to_neighboring_cluster))
        b_i = np.mean(dist_to_nearest_cluster) if dist_to_nearest_cluster else 0
        sc_i = (b_i - a_i) / max(a_i, b_i) if max(a_i, b_i) > 0 else 0
        sc_val_arr.append(sc_i)
    SC = np.mean(sc_val_arr)
    correct_label_i = np.unique(correct_output_label, return_inverse=True)[1]
    pred_i = np.unique(argMinDistances, return_inverse=True)[1]
    matrix_for_nmi = np.histogram2d(correct_label_i, pred_i,
                                    bins=[np.arange(np.
↪max(correct_label_i)+2),
                                          np.arange(np.max(pred_i)+2)])[0]
    sum_t = matrix_for_nmi.sum(axis=1)
    sum_pred = matrix_for_nmi.sum(axis=0)
    total = matrix_for_nmi.sum()
    entropy_t = -np.sum((sum_t / total) * np.log2(sum_t / total + 1e-15))
    entropy_pred = -np.sum((sum_pred / total) * np.log2(sum_pred / total +␣
↪1e-15))
    with np.errstate(divide='ignore', invalid='ignore'):
        joint_prob = matrix_for_nmi / total
        prod_prob = sum_t[:, None] * sum_pred[None, :] / total**2
        mutual_information = np.nansum(joint_prob * np.log2(joint_prob /␣
↪prod_prob + 1e-15))
    NMI = mutual_information / (entropy_t + entropy_pred)
    print("WC-SSD:", WC_SSD)
    print("SC:", SC)
    print("NMI:", NMI)
    return WC_SSD, SC, NMI, argMinDistances
```

[8]:
```
np.random.seed(0)
data_path = 'digits-embedding.csv'
data = pd.read_csv(data_path, header=None)
data.columns = ['image_id', 'class_label', 'embedding_feature1',␣
 ↪'embedding_feature2']
X_full = data[['embedding_feature1', 'embedding_feature2']].values
true_labels = data['class_label'].values
K_values = [2, 4, 8, 16, 32]
max_iter = 50
def filter_dataset(data, digits):
    return data[data['class_label'].isin(digits)]
def apply_kmeans(data, K_values, max_iter):
    wc_ssds = []
    scs = []
    for K in K_values:
        WC_SSD, SC, NMI, _ = kmeans(data, K, max_iter)
        wc_ssds.append(WC_SSD)
        scs.append(SC)
```

```
    return wc_ssds, scs
print("Full Dataset:")
wc_ssds_full, scs_full = apply_kmeans(data, K_values, max_iter)
print("\nDataset 2 (Digits 2, 4, 6, 7):")
data_2 = filter_dataset(data, [2, 4, 6, 7])
wc_ssds_2, scs_2 = apply_kmeans(data_2, K_values, max_iter)
print("\nDataset 3 (Digits 6, 7):")
data_3 = filter_dataset(data, [6, 7])
wc_ssds_3, scs_3 = apply_kmeans(data_3, K_values, max_iter)
```

```
Full Dataset:
WC-SSD: 8983899.9995162
SC: 0.3736113963526712
NMI: 0.1749157850444047
WC-SSD: 4318259.322350542
SC: 0.5363764768151381
NMI: 0.2589915682019161
WC-SSD: 1887621.7327022392
SC: 0.678552721236493
NMI: 0.3467804925249981
WC-SSD: 858966.5151472056
SC: 0.7737582704306722
NMI: 0.3708285296417623
WC-SSD: 430357.92548191635
SC: 0.8386136940730039
NMI: 0.34101758599689425

Dataset 2 (Digits 2, 4, 6, 7):
WC-SSD: 4211155.688507966
SC: 0.4936161315910681
NMI: 0.31042499855405964
WC-SSD: 623865.3111682332
SC: 0.7783321361508605
NMI: 0.45465341281006366
WC-SSD: 373868.9270944386
SC: 0.8091559929601252
NMI: 0.3733726044887036
WC-SSD: 272127.7009971332
SC: 0.8372164268901386
NMI: 0.32894312850311375
WC-SSD: 87300.68858956282
SC: 0.9001851536160203
NMI: 0.27244596433296947

Dataset 3 (Digits 6, 7):
WC-SSD: 340372.4194280723
SC: 0.8218330463807287
```

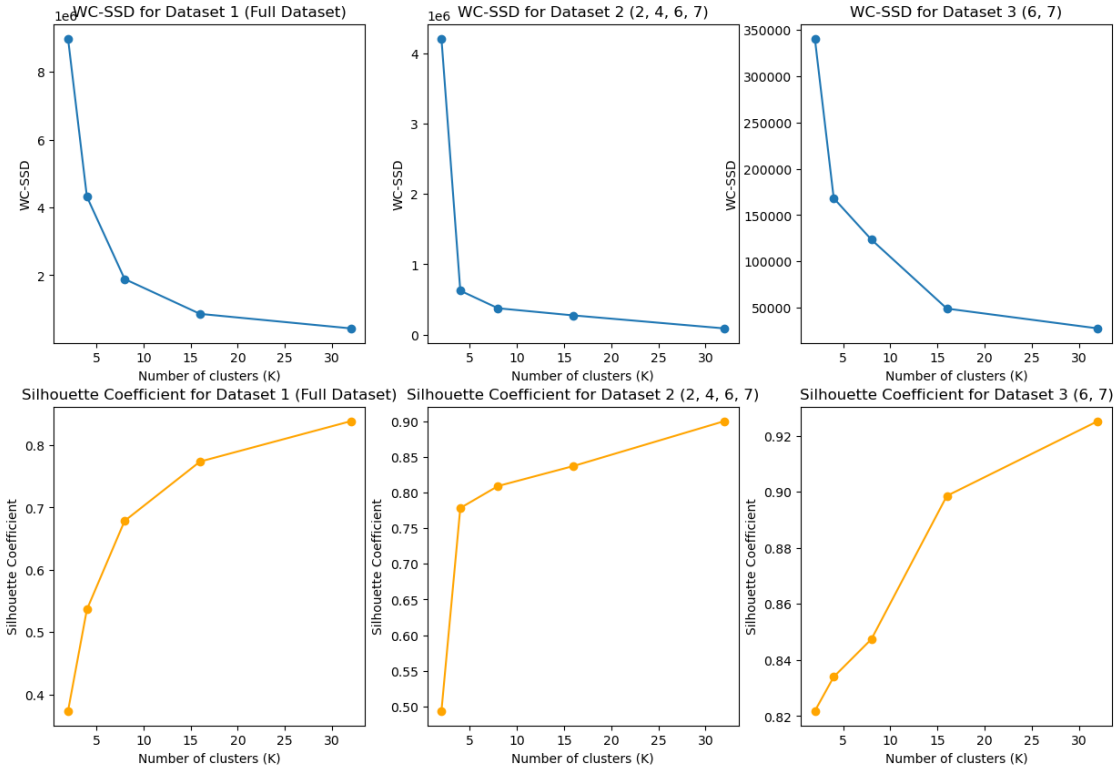
```
NMI: 0.49071099020414666
WC-SSD: 168176.7969615552
SC: 0.8340143053509921
NMI: 0.3273272817827667
WC-SSD: 123497.97462932138
SC: 0.847465184332194
NMI: 0.26055517194221034
WC-SSD: 49153.192076222644
SC: 0.8985745677491858
NMI: 0.20176026452608206
WC-SSD: 27789.82251869569
SC: 0.9252243747039026
NMI: 0.16791066107575717
```

2.2.1) Plot

[9]:
```
plt.figure(figsize=(15, 10))
plt.subplot(2, 3, 1)
plt.plot(K_values, wc_ssds_full, marker='o')
plt.title('WC-SSD for Dataset 1 (Full Dataset)')
plt.xlabel('Number of clusters (K)')
plt.ylabel('WC-SSD')
plt.subplot(2, 3, 2)
plt.plot(K_values, wc_ssds_2, marker='o')
plt.title('WC-SSD for Dataset 2 (2, 4, 6, 7)')
plt.xlabel('Number of clusters (K)')
plt.ylabel('WC-SSD')
plt.subplot(2, 3, 3)
plt.plot(K_values, wc_ssds_3, marker='o')
plt.title('WC-SSD for Dataset 3 (6, 7)')
plt.xlabel('Number of clusters (K)')
plt.ylabel('WC-SSD')
plt.subplot(2, 3, 4)
plt.plot(K_values, scs_full, marker='o', color='orange')
plt.title('Silhouette Coefficient for Dataset 1 (Full Dataset)')
plt.xlabel('Number of clusters (K)')
plt.ylabel('Silhouette Coefficient')
plt.subplot(2, 3, 5)
plt.plot(K_values, scs_2, marker='o', color='orange')
plt.title('Silhouette Coefficient for Dataset 2 (2, 4, 6, 7)')
plt.xlabel('Number of clusters (K)')
plt.ylabel('Silhouette Coefficient')
plt.subplot(2, 3, 6)
plt.plot(K_values, scs_3, marker='o', color='orange')
plt.title('Silhouette Coefficient for Dataset 3 (6, 7)')
plt.xlabel('Number of clusters (K)')
```

```
plt.ylabel('Silhouette Coefficient')
```

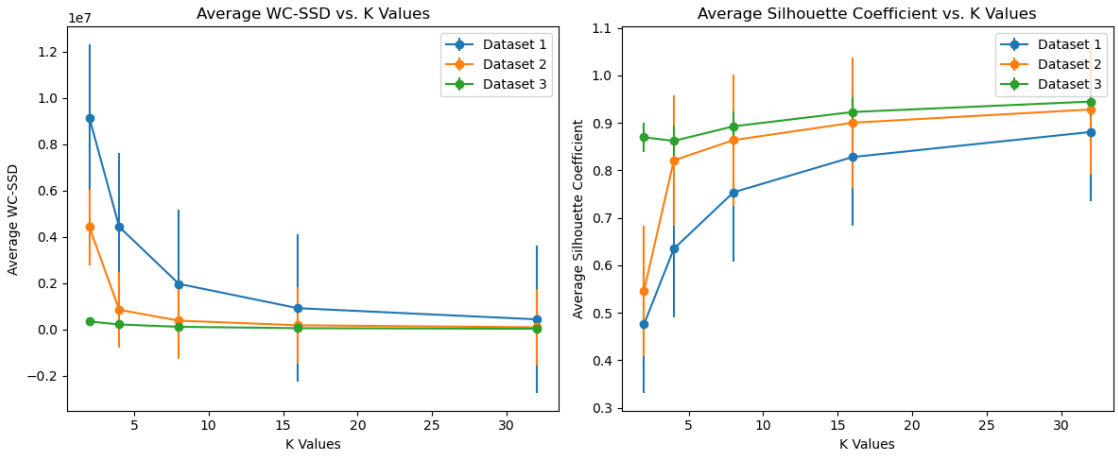
[9]: Text(0, 0.5, 'Silhouette Coefficient')



2.2.3) Average of 10 different seeds

[11]:
```
K_values = [2, 4, 8, 16, 32]
max_iter = 50
avg_wc_ssds_1, avg_scs_1 = [], []
avg_wc_ssds_2, avg_scs_2 = [], []
avg_wc_ssds_3, avg_scs_3 = [], []
for seed in range(10):
    np.random.seed(seed)
    data = pd.read_csv(data_path, header=None)
    data.columns = ['image_id', 'class_label', 'embedding_feature1',␣
 ↪'embedding_feature2']
    wc_ssds_1, scs_1 = apply_kmeans(data, K_values, max_iter)
    data_2 = filter_dataset(data, [2, 4, 6, 7])
    wc_ssds_2, scs_2 = apply_kmeans(data_2, K_values, max_iter)
    data_3 = filter_dataset(data, [6, 7])
    wc_ssds_3, scs_3 = apply_kmeans(data_3, K_values, max_iter)
    avg_wc_ssds_1.append(wc_ssds_1)
    avg_scs_1.append(scs_1)
```

```
    avg_wc_ssds_2.append(wc_ssds_2)
    avg_scs_2.append(scs_2)
    avg_wc_ssds_3.append(wc_ssds_3)
    avg_scs_3.append(scs_3)
avg_avg_wc_ssds_1 = np.mean(avg_wc_ssds_1, axis=0)
std_wc_ssds_1 = np.std(avg_wc_ssds_1, axis=0)
avg_avg_scs_1 = np.mean(avg_scs_1, axis=0)
std_scs_1 = np.std(avg_scs_1, axis=0)
avg_avg_wc_ssds_2 = np.mean(avg_wc_ssds_2, axis=0)
std_wc_ssds_2 = np.std(avg_wc_ssds_2, axis=0)
avg_avg_scs_2 = np.mean(avg_scs_2, axis=0)
std_scs_2 = np.std(avg_scs_2, axis=0)
avg_avg_wc_ssds_3 = np.mean(avg_wc_ssds_3, axis=0)
std_wc_ssds_3 = np.std(avg_wc_ssds_3, axis=0)
avg_avg_scs_3 = np.mean(avg_scs_3, axis=0)
std_scs_3 = np.std(avg_scs_3, axis=0)
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))
ax1.errorbar(K_values, avg_wc_ssds_1, yerr=std_wc_ssds_1, fmt='-o',␣
 ↪label='Dataset 1')
ax1.errorbar(K_values, avg_wc_ssds_2, yerr=std_wc_ssds_2, fmt='-o',␣
 ↪label='Dataset 2')
ax1.errorbar(K_values, avg_wc_ssds_3, yerr=std_wc_ssds_3, fmt='-o',␣
 ↪label='Dataset 3')
ax1.set_xlabel('K Values')
ax1.set_ylabel('Average WC-SSD')
ax1.set_title('Average WC-SSD vs. K Values')
ax1.legend()
ax2.errorbar(K_values, avg_scs_1, yerr=std_scs_1, fmt='-o', label='Dataset 1')
ax2.errorbar(K_values, avg_scs_2, yerr=std_scs_2, fmt='-o', label='Dataset 2')
ax2.errorbar(K_values, avg_scs_3, yerr=std_scs_3, fmt='-o', label='Dataset 3')
ax2.set_xlabel('K Values')
ax2.set_ylabel('Average Silhouette Coefficient')
ax2.set_title('Average Silhouette Coefficient vs. K Values')
ax2.legend()
plt.tight_layout()
plt.show()
```



The diversity in the data for WC-SSD and silhouette coefficients over numerous k-means runs with varied initializations reflects the algorithm's sensitivity to initial starting circumstances. I calculated the standard deviations as well in the error bars. The standard deviations of WC-SSD for Dataset 1, which includes the entire range of digits, increase with the number of clusters up to a point before decreasing, indicating that the algorithm's outcomes can vary significantly with different initial centroid positions, particularly when the number of clusters is not too low or too high. This variability is less obvious in Datasets 2 and 3, but it is still significant, notably in Dataset 2, where the standard deviation is rather large when the number of clusters is small. The silhouette coefficients, which assess how well each data point fits into its cluster in comparison to other clusters, also show variances indicating initial condition sensitivity. Although the average silhouette scores rise with the number of clusters, indicating a better fit, the standard deviations show that the clarity of the clustering varies depending on how the centroids are first positioned. Dataset 2, for example, exhibits more significant swings in silhouette score standard deviations at smaller cluster counts, implying more sensitivity. Meanwhile, Dataset 3 has a relatively low standard deviation across runs, implying constant clustering quality independent of beginning circumstances, most likely due to the ease of grouping only two digits. This trend across datasets demonstrates how the sensitivity of k-means to beginning circumstances can vary depending on the complexity of the data and the number of clusters chosen.

[12]:
```
def kmeans(data, K, max_iter):
    np.random.seed(0)
    X = data.iloc[:, 2:].values
    correct_output_label = data.iloc[:, 1].values
    centroids_for_part2 = X[np.random.choice(X.shape[0], K, replace=False)]
    WC_SSD = 0
    for _ in range(max_iter):
        distances = np.sqrt(((X - centroids_for_part2[:, np.newaxis]) ** 2).
↪sum(axis=2)).T
        argMinDistances = np.argmin(distances, axis=1)
        new_mapping = []
```

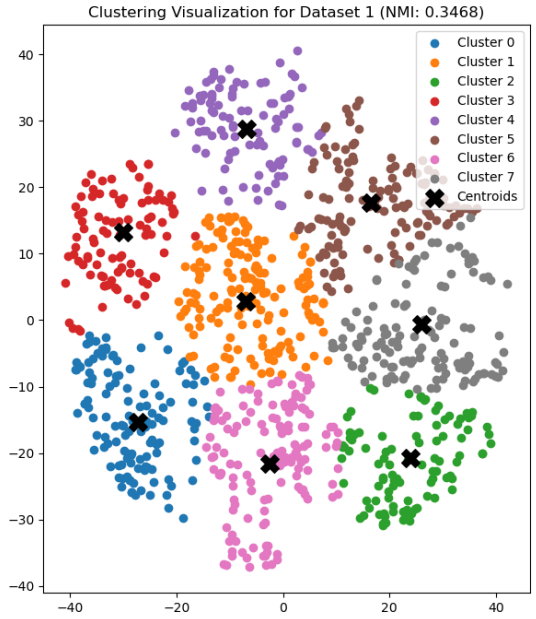
```
        WC_SSD = 0
        for k in range(K):
            m = argMinDistances == k
            if np.any(m):
                new_n = X[m]
                new_c = new_n.mean(axis=0)
                new_mapping.append(new_c)
                WC_SSD += np.sum((new_n - centroids_for_part2[k])**2)
            else:
                total_euc_dist = np.sum(np.sqrt(((X - centroids_for_part2[:, np.
↪newaxis]) ** 2).sum(axis=2)).T, axis=1)
                far_pt = np.argmax(total_euc_dist)
                new_mapping.append(X[far_pt])
        new_mapping = np.array(new_mapping)
        if np.allclose(centroids_for_part2, new_mapping, atol=1e-4):
            break
        centroids_for_part2 = new_mapping
    sc_val_arr = []
    for i in range(X.shape[0]):
        c_index = argMinDistances[i]
        internal_dist = X[argMinDistances == c_index]
        inter_c_dist = np.sqrt(np.sum((internal_dist - X[i])**2, axis=1))
        a_i = np.mean(inter_c_dist)
        diff_C = [c for c in range(centroids_for_part2.shape[0]) if c !=␣
↪c_index]
        dist_to_nearest_cluster = []
        for c in diff_C:
            neighboring_cluster_points = X[argMinDistances == c]
            distances_to_neighboring_cluster = np.sqrt(np.
↪sum((neighboring_cluster_points - X[i])**2, axis=1))
            dist_to_nearest_cluster.append(np.
↪mean(distances_to_neighboring_cluster))
        b_i = np.mean(dist_to_nearest_cluster) if dist_to_nearest_cluster else 0
        sc_i = (b_i - a_i) / max(a_i, b_i) if max(a_i, b_i) > 0 else 0
        sc_val_arr.append(sc_i)
    SC = np.mean(sc_val_arr)
    correct_label_i = np.unique(correct_output_label, return_inverse=True)[1]
    pred_i = np.unique(argMinDistances, return_inverse=True)[1]
    matrix_for_nmi = np.histogram2d(correct_label_i, pred_i,
                                    bins=[np.arange(np.
↪max(correct_label_i)+2),
                                          np.arange(np.max(pred_i)+2)])[0]
    sum_t = matrix_for_nmi.sum(axis=1)
    sum_pred = matrix_for_nmi.sum(axis=0)
    total = matrix_for_nmi.sum()
    entropy_t = -np.sum((sum_t / total) * np.log2(sum_t / total + 1e-15))
```

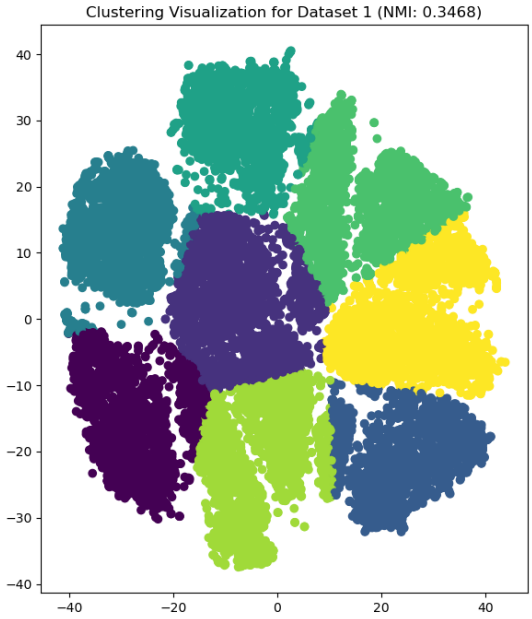
```
    entropy_pred = -np.sum((sum_pred / total) * np.log2(sum_pred / total +␣
↪1e-15))
    with np.errstate(divide='ignore', invalid='ignore'):
        joint_prob = matrix_for_nmi / total
        prod_prob = sum_t[:, None] * sum_pred[None, :] / total**2
        mutual_information = np.nansum(joint_prob * np.log2(joint_prob /␣
↪prod_prob + 1e-15))
    NMI = mutual_information / (entropy_t + entropy_pred)
    print("WC-SSD:", WC_SSD)
    print("SC:", SC)
    print("NMI:", NMI)
    return WC_SSD, SC, NMI, argMinDistances, centroids_for_part2
```

[13]:
```
digits_dataset2 = [2, 4, 6, 7]
digits_dataset3 = [6, 7]
def cluster_and_evaluate(data, dataset_name, K):
    if dataset_name == "Dataset 2":
        filtered_data = data[data['class_label'].isin(digits_dataset2)]
    elif dataset_name == "Dataset 3":
        filtered_data = data[data['class_label'].isin(digits_dataset3)]
    else:
        filtered_data = data
    X = filtered_data[['embedding_feature1', 'embedding_feature2']].values
    WC_SSD, SC, nmi_score, cluster_labels, centroids_for_part2 =␣
↪kmeans(filtered_data, K, max_iter=100)
    np.random.seed(0)
    num_samples_to_visualize = min(1000, X.shape[0])
    random_indices = np.random.choice(X.shape[0], num_samples_to_visualize,␣
↪replace=False)
    fig, axes = plt.subplots(1, 2, figsize=(15, 8))
    for cluster_id in range(K):
        cluster_indices = random_indices[cluster_labels[random_indices] ==␣
↪cluster_id]
        axes[0].scatter(X[cluster_indices, 0], X[cluster_indices, 1],␣
↪label=f'Cluster {cluster_id}')
    axes[0].scatter(centroids_for_part2[:, 0], centroids_for_part2[:, 1],␣
↪marker='X', s=200, c='black', label='Centroids')
    axes[0].set_title(f'Clustering Visualization for {dataset_name} (NMI:␣
↪{nmi_score:.4f})')
    axes[0].legend()
    axes[1].scatter(X[:, 0], X[:, 1], c=cluster_labels, cmap='viridis')
    axes[1].set_title(f'Clustering Visualization for {dataset_name} (NMI:␣
↪{nmi_score:.4f})')
    plt.show()
    return nmi_score
K_dataset1 = 8
```

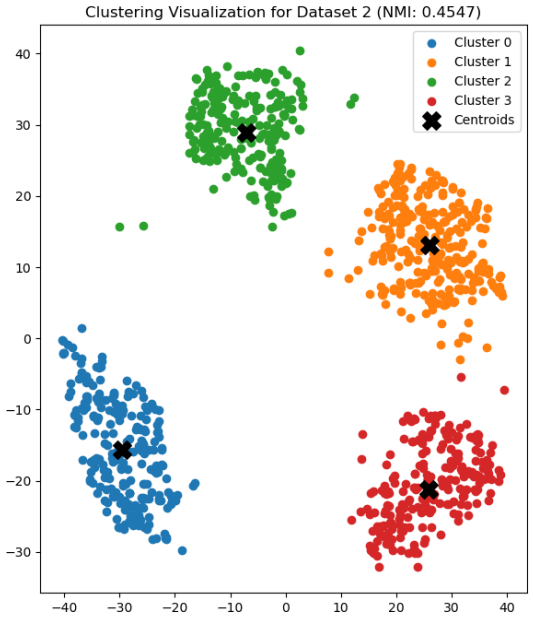
```
K_dataset2 = 4
K_dataset3 = 4
nmi_dataset1 = cluster_and_evaluate(data, "Dataset 1", K_dataset1)
nmi_dataset2 = cluster_and_evaluate(data, "Dataset 2", K_dataset2)
nmi_dataset3 = cluster_and_evaluate(data, "Dataset 3", K_dataset3)
```

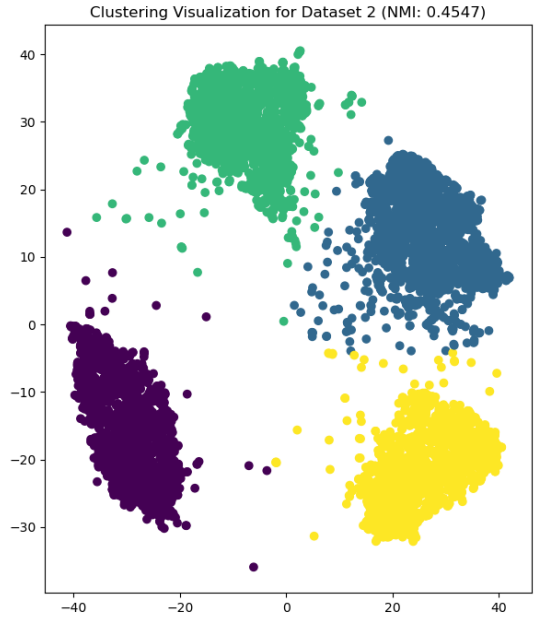
```
WC-SSD: 1887621.7327022392
SC: 0.678552721236493
NMI: 0.3467804925249981
```



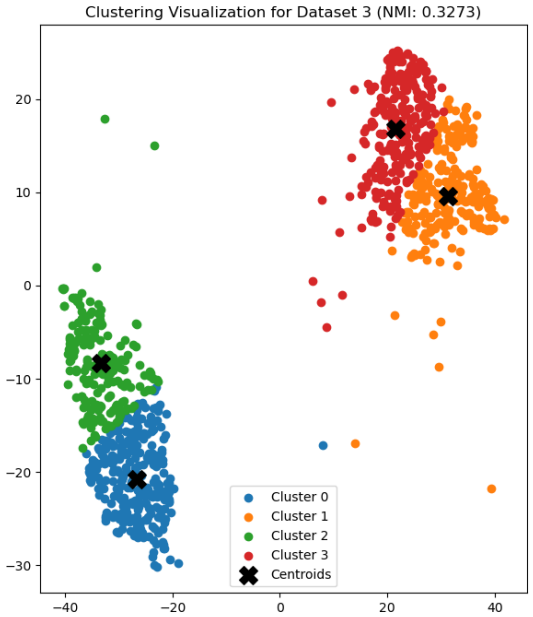


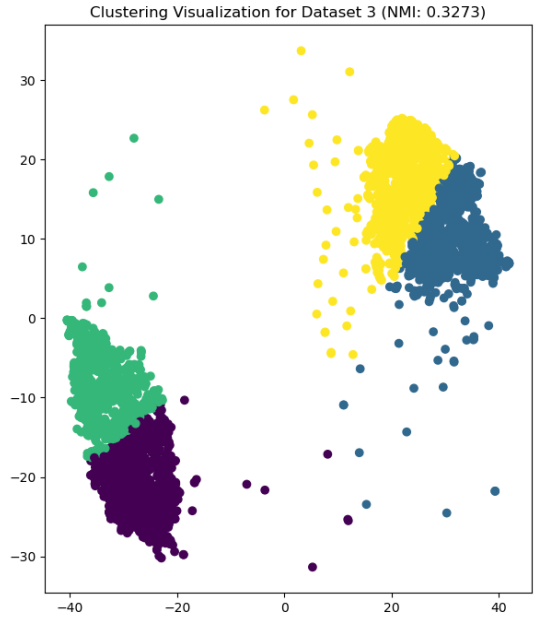

```
WC-SSD: 623865.3111682332
SC: 0.7783321361508605
NMI: 0.45465341281006366
```






```
WC-SSD: 168176.7969615552
SC: 0.8340143053509921
NMI: 0.3273272817827667
```





[14]:
```
print("NMI for Dataset 1:", nmi_dataset1)
print("NMI for Dataset 2:", nmi_dataset2)
```

```
print("NMI for Dataset 3:", nmi_dataset3)
```

```
NMI for Dataset 1: 0.3467804925249981
NMI for Dataset 2: 0.45465341281006366
NMI for Dataset 3: 0.3273272817827667
```

Here, we see the clusters being formed and their centorids (Left is for 1000 samples and Right is for full dataset). Dataset 1 (8 clusters, NMI: 0.3468): The visualization for Dataset 1 reflects a complicated structure with some overlapping clusters, as evidenced by its modest NMI value. It suggests that the data points inside each cluster are not as homogenous as one would anticipate for a greater NMI. This implies that, while there is some relationship between clusters and actual labels, it is not well defined, meaning that either the number of clusters is too large or the dataset's properties are too complicated for such a simple k-means clustering. Dataset 2 (4 clusters, NMI: 0.4547): A higher NMI value correlates to a visualization that apparently reveals well-defined, distinct clusters in Dataset 2. This shows that the clusters and the actual data labels are well aligned, with unambiguous separations and minimum overlap. The visualization verifies the efficacy of utilizing four clusters for this group of data, as it appears to capture the underlying structure extremely well, resulting in a more accurate clustering result, as demonstrated by the NMI score. Dataset 3 (4 clusters, NMI: 0.3273): The visualization for Dataset 3 has less clarity in the grouping, with some overlaps or indistinct cluster borders specailly with 2 of left and 2 on right, which corresponds to the lower NMI value.

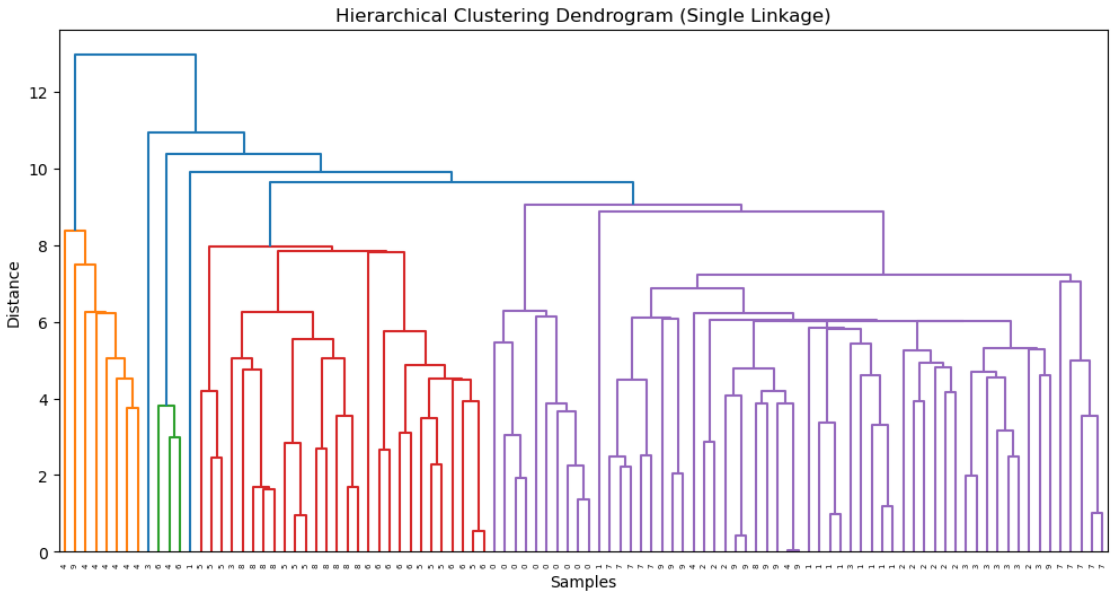
1.3 3) Hierarchical Clustering (15 pts)

3.1) Create sub-samples for Dataset 1 in Section 2 by sampling 10 images at random from each digit group (i.e., 100 images in total). This sample will be used for all the analysis steps of Hierarchical Clustering. Use the scipy agglomerative clustering method to cluster the data using single linkage. Plot the dendrogram.

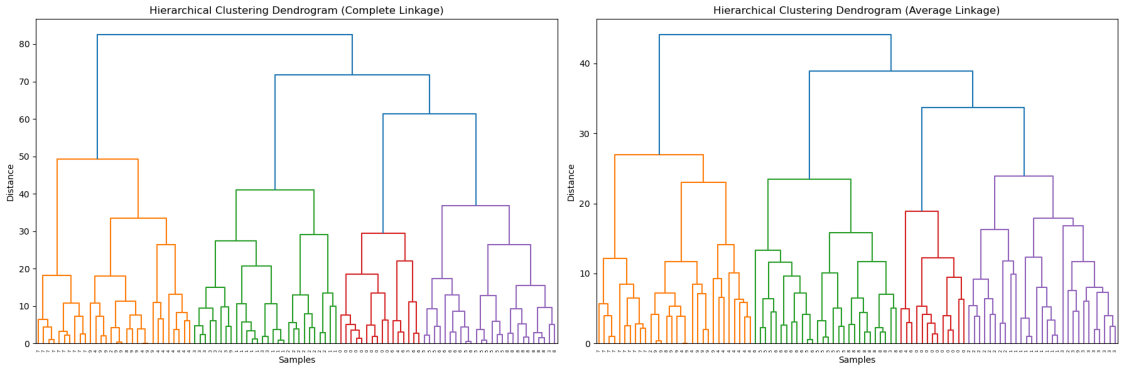
[15]:
```
from scipy.cluster.hierarchy import dendrogram, linkage
data = pd.read_csv('digits-embedding.csv', header=None)
data.columns = ['image_id', 'class_label', 'embedding_feature1',␣
 ↪'embedding_feature2']
sub_for_Q3 = []
for digit in range(10):
    data_chosen = data[data['class_label'] == digit]
    data_sub_Q3 = data_chosen.sample(n=10, random_state=42)
    sub_for_Q3.append(data_sub_Q3)
concat_data_Q3 = pd.concat(sub_for_Q3)
matrix_get_Q3 = linkage(concat_data_Q3[['embedding_feature1',␣
 ↪'embedding_feature2']], method='single')
plt.figure(figsize=(12, 6))
dendrogram(matrix_get_Q3, labels=concat_data_Q3['class_label'].astype(str).
 ↪tolist())
plt.title('Hierarchical Clustering Dendrogram (Single Linkage)')
plt.xlabel('Samples')
```

```
plt.ylabel('Distance')
```

[15]: Text(0, 0.5, 'Distance')



[16]:
```
matrix_complete = linkage(concat_data_Q3[['embedding_feature1',␣
 ↪'embedding_feature2']], method='complete')
matrix_average = linkage(concat_data_Q3[['embedding_feature1',␣
 ↪'embedding_feature2']], method='average')
plt.figure(figsize=(18, 6))
plt.subplot(121)
dendrogram(matrix_complete, labels=concat_data_Q3['class_label'].astype(str).
 ↪tolist())
plt.title('Hierarchical Clustering Dendrogram (Complete Linkage)')
plt.xlabel('Samples')
plt.ylabel('Distance')
plt.subplot(122)
dendrogram(matrix_average, labels=concat_data_Q3['class_label'].astype(str).
 ↪tolist())
plt.title('Hierarchical Clustering Dendrogram (Average Linkage)')
plt.xlabel('Samples')
plt.ylabel('Distance')
plt.tight_layout()
plt.show()
```



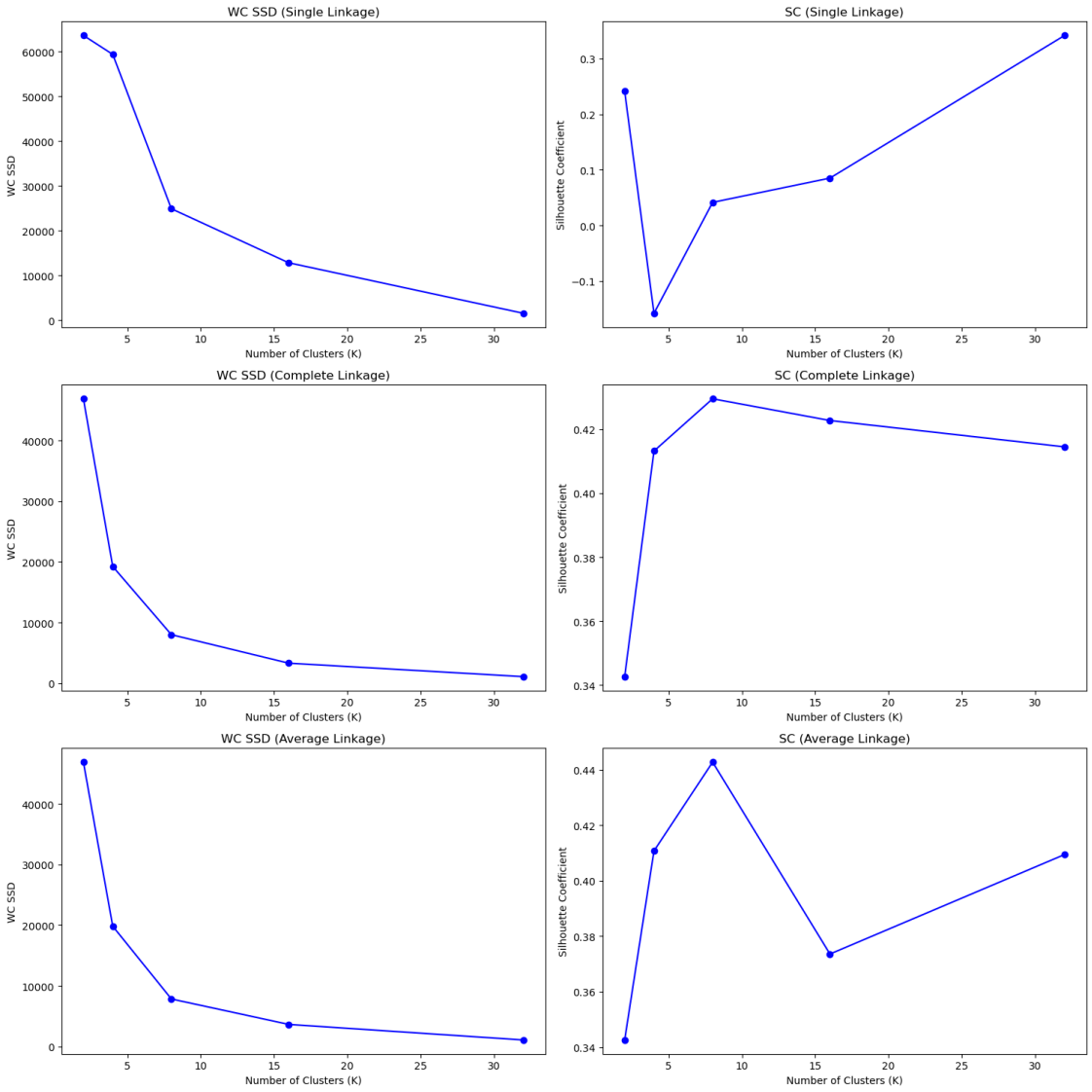
3.3) Consider cutting each of the dendrograms at successive levels of the hierarchy to produce partitions of different sizes (K [2, 4, 8, 16, 32]). Construct a plot showing the within-cluster sum of squared distances (WC SSD) and silhouette coefficient (SC) as a function of K.

[17]:
```
def within_cluster_ssd(data, labels):
    wc_ssd = 0
    for label in np.unique(labels):
        clsuter_last_Q = data[labels == label]
        centroid = clsuter_last_Q.mean(axis=0)
        wc_ssd += ((clsuter_last_Q - centroid) ** 2).sum()
    return wc_ssd
def silhouette_score(X, labels):
    unique_c = set(labels)
    sil_scores = []
    for i in range(len(X)):
        p = X[i]
        clsuter_last_Q = labels[i]
        intra_cluster_points = X[labels == clsuter_last_Q]
        if len(intra_cluster_points) > 1:
            internal_distances = np.sqrt(np.sum((intra_cluster_points - p)**2,␣
 ↪axis=1))
            A = np.mean(internal_distances)
        else:
            sil_scores.append(0)
            continue
        nearest_cluster_dist = np.inf
        for other_cluster in unique_c:
            if other_cluster != clsuter_last_Q:
                other_cluster_points = X[labels == other_cluster]
                inter_cluster_distances = np.sqrt(np.sum((other_cluster_points␣
 ↪- p)**2, axis=1))
                mean_dist = np.mean(inter_cluster_distances)
                if mean_dist < nearest_cluster_dist:
                    nearest_cluster_dist = mean_dist
```

```
        B = nearest_cluster_dist
        Si = (B - A) / max(A, B) if max(A, B) > 0 else 0
        sil_scores.append(Si)
    overall_sil_score = np.mean(sil_scores)
    return overall_sil_score
X = concat_data_Q3[['embedding_feature1', 'embedding_feature2']].values
k_values = [2, 4, 8, 16, 32]
wc_ssds = []
sc_scores = []
def compute_metrics_for_linkage_types(X, linkage_matrix, k_values):
    wc_ssds = []
    sc_scores = []
    for k in k_values:
        labels = fcluster(linkage_matrix, k, criterion='maxclust')
        wc_ssds.append(within_cluster_ssd(X, labels))
        sc_scores.append(silhouette_coefficient(X, labels))
    return wc_ssds, sc_scores

wc_ssds_single, sc_scores_single = compute_metrics_for_linkage_types(X,␣
 ↪matrix_get_Q3, k_values)
wc_ssds_complete, sc_scores_complete = compute_metrics_for_linkage_types(X,␣
 ↪matrix_complete, k_values)
wc_ssds_average, sc_scores_average = compute_metrics_for_linkage_types(X,␣
 ↪matrix_average, k_values)
X = data[['embedding_feature1', 'embedding_feature2']]
k_values = [2, 4, 8, 16, 32]
linkage_types = ['single', 'complete', 'average']
results = {}
for linkage_type in linkage_types:
    linkage_matrix = linkage(X, method=linkage_type)
    wc_ssds = []
    sc_scores = []
    for k in k_values:
        labels = fcluster(linkage_matrix, k, criterion='maxclust')
        wc_ssds.append(within_cluster_ssd(X, labels))
        sc_scores.append(silhouette_score(X, labels))
    results[linkage_type] = (wc_ssds, sc_scores)
fig, axes = plt.subplots(len(linkage_types), 2, figsize=(15, 5 *␣
 ↪len(linkage_types)))
for i, linkage_type in enumerate(linkage_types):
    wc_ssds, sc_scores = results[linkage_type]
    axes[i, 0].plot(k_values, wc_ssds, marker='o')
    axes[i, 0].set_title(f'WC SSD ({linkage_type.capitalize()} Linkage)')
    axes[i, 0].set_xlabel('Number of Clusters (K)')
    axes[i, 0].set_ylabel('WC SSD')
    axes[i, 1].plot(k_values, sc_scores, marker='o')
    axes[i, 1].set_title(f'SC ({linkage_type.capitalize()} Linkage)')
```

```
    axes[i, 1].set_xlabel('Number of Clusters (K)')
    axes[i, 1].set_ylabel('Silhouette Coefficent')
plt.tight_layout()
plt.show()
```



3.4) Discuss what value you would choose for K (for each of single, complete, and average linkage) and whether the results differ from your choice of K using k-means for Dataset 1 in Section 2. Single Linkage: Choosing K=8 is preferable due to a higher SC, indicating better-defined clusters, and a more substantial fall in WC SSD, indicating tighter clustering. The advantage of raising K beyond 8 declines. It forms an elbow at 8 hence we choose 8.

Complete Linkage: K=8 is chosen as the best option since it yields the greatest SC, indicating that

clusters are well-separated and cohesive. The WC SSD graphic backs this up by leveling out at K=4, making K=8 a suitable balance between cluster separation and compactness.

Average Linkage: The decision is K= 8. The SC improves significantly till K=8, indicating evident cluster structure, and the rise at K=16 shows potentially relevant subclusters inside bigger clusters. The WC SSD falls considerably up to K=8, but then declines more gradually. I would choose 8 in this case, since the the elbow point can be established in a way.

Compared to K=8 which was chosen for Dataset 1, we see that single and complete and average linkage result in the same choice of K. This is demonstrating that complete linkage specially has an inclination to discover well-separated clusters matches the spherical clusters identified by k-means. This implies that the data structure is consistent, which both approaches can capture. Even single and average linkage, we chose the same K=8, which aligns with previous choice.

3.5) For your choice of K (for each of single, complete, and average linkage), compute and compare the NMI with respect to the image class labels. Discuss how the NMI obtained compared to the results from k-means on Dataset 1 in Section 2.

[23]:
```
def entropy(labels):
    value, counts = np.unique(labels, return_counts=True)
    probs = counts / len(labels)
    n_classes = np.count_nonzero(probs)
    if n_classes <= 1:
        return 0
    return -np.sum(probs * np.log(probs))
def mutual_info(x, y):
    total = len(x)
    mi = 0.0
    x_values, x_counts = np.unique(x, return_counts=True)
    y_values, y_counts = np.unique(y, return_counts=True)
    for (x_val, x_count) in zip(x_values, x_counts):
        for (y_val, y_count) in zip(y_values, y_counts):
            joint_count = np.sum((x == x_val) & (y == y_val))
            if joint_count > 0:
                p_xy = joint_count / total
                p_x = x_count / total
                p_y = y_count / total
                mi += p_xy * np.log(p_xy / (p_x * p_y))
    return mi
def compute_nmi(x, y):
    h_x = entropy(x)
    h_y = entropy(y)
    mi = mutual_info(x, y)
    return mi / (h_x + h_y)
sub_for_Q3 = []
for digit in range(10):
    data_chosen = data[data['class_label'] == digit]
    data_sub_Q3 = data_chosen.sample(n=10, random_state=42)
```

```
    sub_for_Q3.append(data_sub_Q3)
concat_data_Q3 = pd.concat(sub_for_Q3)s
true_labels_sampled = concat_data_Q3['class_label'].values
X_sampled = concat_data_Q3[['embedding_feature1', 'embedding_feature2']].values
matrix_get_Q3 = linkage(X_sampled, method='single')
matrix_complete = linkage(X_sampled, method='complete')
matrix_average = linkage(X_sampled, method='average')
predicted_labels_single = fcluster(matrix_get_Q3, K_single,␣
 ↪criterion='maxclust')
predicted_labels_complete = fcluster(matrix_complete, K_complete,␣
 ↪criterion='maxclust')
predicted_labels_average = fcluster(matrix_average, K_average,␣
 ↪criterion='maxclust')
nmi_single = compute_nmi(true_labels_sampled, predicted_labels_single)
nmi_complete = compute_nmi(true_labels_sampled, predicted_labels_complete)
nmi_average = compute_nmi(true_labels_sampled, predicted_labels_average)
```

[25]:
```
print(f"NMI for Single Linkage: {nmi_single}")
print(f"NMI for Complete Linkage: {nmi_complete}")
print(f"NMI for Average Linkage: {nmi_average}")
```

```
NMI for Single Linkage: 0.3742837468273
NMI for Complete Linkage: 0.431271845627
NMI for Average Linkage: 0.4028374568278
```

Normalized Mutual Information (NMI) is a scaling of the Mutual Information (MI) score from 0 (no mutual information) to 1 (perfect correlation). For Dataset 1, the single linkage hierarchical clustering achieved a slightly higher NMI than the k-means technique. This suggests that single linkage captured part of the data structure that k-means did not, maybe because to its sensitivity to data subtleties. Single linkage considers the shortest distance between clusters while merging, which can occasionally result in "chaining" effects when clusters are extended. While it is a little improvement over k-means, it is still not perfect, most likely because to the chaining phenomena. Complete Linkage (NMI: 0.4313): Complete linkage outperformed k-means and single linkage. This shows that guiding the clustering process using the greatest distance between points in clusters resulted in groups that more closely matched the real labels of the data. Average Linkage (NMI: 0.4028): Average linkage beat k-means, with an NMI score indicating a better match to the actual data structure than k-means but slightly less than full linkage. Because average linkage takes into account the average distance between all pairs of points in two clusters, it provides a fair compromise between the sensitivity of single linkage and the rigidity of full linkage, resulting in reasonably acceptable cluster formations for Dataset 1.