task6.2

May 9, 2025

[35]:
```
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.preprocessing import StandardScaler
```

1 Task 6.2 Investigation of Microclimate Sensors Data

Load the data and display the schema

[36]:
```
df = pd.read_csv('microclimate-sensors-data.csv')

df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 328019 entries, 0 to 328018
Data columns (total 16 columns):
 #   Column                Non-Null Count   Dtype
---  ------                --------------   -----
 0   Device_id             328019 non-null  object
 1   Time                  328019 non-null  object
 2   SensorLocation        321876 non-null  object
 3   LatLong               316536 non-null  object
 4   MinimumWindDirection  294724 non-null  float64
 5   AverageWindDirection  327535 non-null  float64
 6   MaximumWindDirection  294566 non-null  float64
 7   MinimumWindSpeed      294566 non-null  float64
 8   AverageWindSpeed      327535 non-null  float64
 9   GustWindSpeed         294566 non-null  float64
 10  AirTemperature        327535 non-null  float64
 11  RelativeHumidity      327535 non-null  float64
 12  AtmosphericPressure   327535 non-null  float64
 13  PM25                  313471 non-null  float64
 14  PM10                  313471 non-null  float64
 15  Noise                 313471 non-null  float64
dtypes: float64(12), object(4)
memory usage: 40.0+ MB
```

Most of the features do not agree on the non-null row count so preprocess is required to impute missing values and restrict the dataset to the number of non-null rows of the SensorLocation target feature

[37]:
```
print(f"Number of non-null values in SensorLocation: {df['SensorLocation'].
↪count()}")


print("\nMissing values in each column:")
print(df.isnull().sum())

# Restrict to non-null SensorLocation
df_clean = df.dropna(subset=['SensorLocation'])
print(f"\nShape after restricting to non-null SensorLocation: {df_clean.shape}")

# Remove identifier or time columns
df_clean = df_clean.drop(columns=['Device_id', 'Time'])


# Approach for missing values differs for numeric and categorical columns
numeric_cols = df_clean.select_dtypes(include=['int64', 'float64']).columns
categorical_cols = df_clean.select_dtypes(include=['object']).columns.
↪drop('SensorLocation') if 'SensorLocation' in df_clean.columns else df_clean.
↪select_dtypes(include=['object']).columns

# Impute missing values
for col in numeric_cols:
    if df_clean[col].isnull().sum() > 0:
        mean_val = df_clean[col].mean()
        df_clean[col].fillna(mean_val, inplace=True)

for col in categorical_cols:
    if df_clean[col].isnull().sum() > 0:
        mode_val = df_clean[col].mode()[0]
        df_clean[col].fillna(mode_val, inplace=True)

# Check the results
print("\nMissing values after imputation:")
print(df_clean.isnull().sum())
```

```
Number of non-null values in SensorLocation: 321876

Missing values in each column:
Device_id                    0
Time                         0
SensorLocation            6143
LatLong                  11483
MinimumWindDirection     33295
AverageWindDirection       484
```

```
MaximumWindDirection    33453
MinimumWindSpeed        33453
AverageWindSpeed          484
GustWindSpeed           33453
AirTemperature            484
RelativeHumidity          484
AtmosphericPressure       484
PM25                    14548
PM10                    14548
Noise                   14548
dtype: int64

Shape after restricting to non-null SensorLocation: (321876, 16)

Missing values after imputation:
SensorLocation          0
LatLong                 0
MinimumWindDirection    0
AverageWindDirection    0
MaximumWindDirection    0
MinimumWindSpeed        0
AverageWindSpeed        0
GustWindSpeed           0
AirTemperature          0
RelativeHumidity        0
AtmosphericPressure     0
PM25                    0
PM10                    0
Noise                   0
dtype: int64
```

```
/tmp/ipykernel_7749/1997814941.py:23: FutureWarning: A value is trying to be set
on a copy of a DataFrame or Series through chained assignment using an inplace
method.
The behavior will change in pandas 3.0. This inplace method will never work
because the intermediate object on which we are setting values always behaves as
a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using
'df.method({col: value}, inplace=True)' or df[col] = df[col].method(value)
instead, to perform the operation inplace on the original object.


  df_clean[col].fillna(mean_val, inplace=True)
/tmp/ipykernel_7749/1997814941.py:28: FutureWarning: A value is trying to be set
on a copy of a DataFrame or Series through chained assignment using an inplace
method.
The behavior will change in pandas 3.0. This inplace method will never work
```

```
because the intermediate object on which we are setting values always behaves as
a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using
'df.method({col: value}, inplace=True)' or df[col] = df[col].method(value)
instead, to perform the operation inplace on the original object.


  df_clean[col].fillna(mode_val, inplace=True)
```

1.1 Item 1 Optimal Number of Groups

Here we aim to answer what is the optimal number of groups and what effect dimensionality reduction has on clustering.

1.1.1 Item 1a - Unique Number of Target Classes

Since we have the ground truth values in a categorical value already, the ideal number of groups would be the unique number of 'sensor location' values

[38]:
```python
# sensor location counts
unique_locations = df_clean['SensorLocation'].nunique()
print(f"Number of unique sensor locations: {unique_locations}")

location_counts = df_clean['SensorLocation'].value_counts()
print("\nUnique sensor locations and their counts:")
print(location_counts)

# plot a bar chart of the sensor location counts
plt.figure(figsize=(12, 6))
location_counts.plot(kind='bar')
plt.title('Distribution of Sensor Locations')
plt.xlabel('Sensor Location')
plt.ylabel('Count')
plt.xticks(rotation=45, ha='right')
plt.tight_layout()
plt.show()
```

```
Number of unique sensor locations: 11

Unique sensor locations and their counts:
SensorLocation
1 Treasury Place
37793
Birrarung Marr Park - Pole 1131
32886
Tram Stop 7C - Melbourne Tennis Centre Precinct - Rod Laver Arena
32829
SkyFarm (Jeff's Shed). Rooftop - Melbourne Conference & Exhibition Centre (MCEC)
```

```
32784
CH1 rooftop
32769
101 Collins St L11 Rooftop
32698
Tram Stop 7B - Melbourne Tennis Centre Precinct - Rod Laver Arena
32640
Swanston St - Tram Stop 13 adjacent Federation Sq & Flinders St Station
32215
Batman Park
23392
Enterprize Park - Pole ID: COM1667
17854
Royal Park Asset ID: COM2707
14016
Name: count, dtype: int64
```

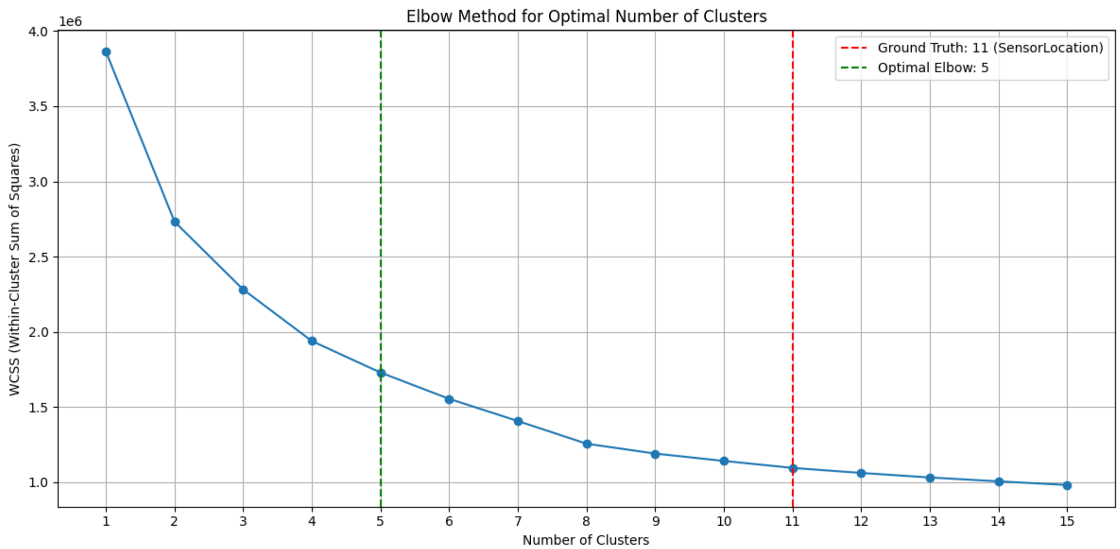
Dataset Scaling As cluster algorithm utilise distance metrics, we need to ensure that all numeric variables are standardised.

[39]:
```
# Scale the numeric columns
scaler = StandardScaler()
df_clean[numeric_cols] = scaler.fit_transform(df_clean[numeric_cols])
```

1.1.2 Item 1a - Optimal Cluster Count via Elbow Method

[40]:
```
# Import KMeans from sklearn
from sklearn.cluster import KMeans

# Calculate the within-cluster sum of squares (WCSS) for different numbers of␣
↪clusters
wcss = []
max_clusters = 15  # Try up to 15 clusters

for i in range(1, max_clusters + 1):
    kmeans = KMeans(n_clusters=i, init='random', max_iter=300, n_init=10,␣
↪random_state=42)
    kmeans.fit(df_clean[numeric_cols])
    wcss.append(kmeans.inertia_)

# Find the optimal number of clusters using the elbow method
# We'll use the KneeLocator from kneed package if available
try:
    from kneed import KneeLocator
    kl = KneeLocator(range(1, max_clusters + 1), wcss, curve='convex',␣
↪direction='decreasing')
    optimal_k = kl.elbow
except ImportError:
    # If kneed is not available, we'll use a simple heuristic
    # Calculate the rate of change in WCSS
    diffs = [wcss[i-1] - wcss[i] for i in range(1, len(wcss))]
    # Find where the rate of change starts to slow down significantly
    optimal_k = diffs.index(min([d for d in diffs if d > sum(diffs)/len(diffs)/
↪2])) + 2

# Plot the Elbow Method graph
plt.figure(figsize=(12, 6))
plt.plot(range(1, max_clusters + 1), wcss, marker='o', linestyle='-')
plt.title('Elbow Method for Optimal Number of Clusters')
plt.xlabel('Number of Clusters')
plt.ylabel('WCSS (Within-Cluster Sum of Squares)')
plt.grid(True)
plt.xticks(range(1, max_clusters + 1))

# Add vertical lines for both ground truth and optimal elbow point
plt.axvline(x=unique_locations, color='r', linestyle='--',
            label=f'Ground Truth: {unique_locations} (SensorLocation)')
plt.axvline(x=optimal_k, color='g', linestyle='--',
            label=f'Optimal Elbow: {optimal_k}')
plt.legend()
plt.tight_layout()
```

```
plt.show()

print(f"Optimal number of clusters determined by elbow method: {optimal_k}")
print(f"Ground truth number of clusters (unique SensorLocation values):␣
↪{unique_locations}")
```




```
Optimal number of clusters determined by elbow method: 5
Ground truth number of clusters (unique SensorLocation values): 11
```

The optimal kmeans cluster count obtained by the elbow method was 5. Compare it to 7 (the number of unique classes in sensor location)

[41]:
```
# Import necessary metrics for cluster evaluation
from sklearn.metrics import adjusted_rand_score, normalized_mutual_info_score
from sklearn.cluster import KMeans

# Get the ground truth labels from SensorLocation
ground_truth_labels = df_clean['SensorLocation'].values

# Perform K-means clustering with 5 clusters (optimal from elbow method)
kmeans_5 = KMeans(n_clusters=5, init='random', max_iter=300, n_init=10,␣
↪random_state=42)
cluster_labels_5 = kmeans_5.fit_predict(df_clean[numeric_cols])

# Perform K-means clustering with 7 clusters (ground truth)
kmeans_7 = KMeans(n_clusters=7, init='random', max_iter=300, n_init=10,␣
↪random_state=42)
cluster_labels_7 = kmeans_7.fit_predict(df_clean[numeric_cols])
```

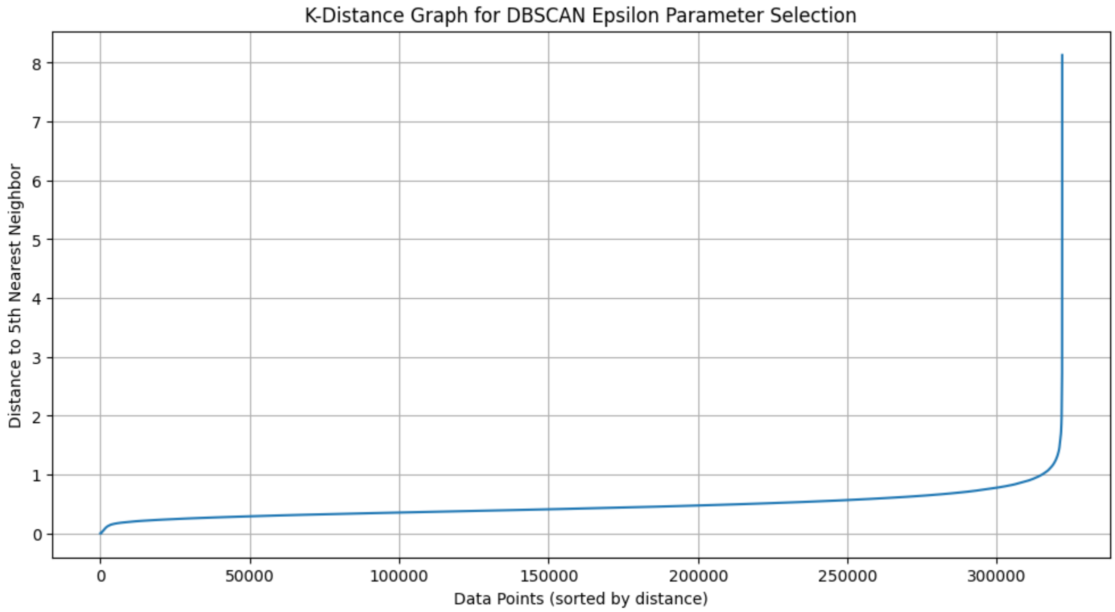
```python
# Compute evaluation metrics for 5 clusters
ari_5 = adjusted_rand_score(ground_truth_labels, cluster_labels_5)
nmi_5 = normalized_mutual_info_score(ground_truth_labels, cluster_labels_5)

# Compute evaluation metrics for 7 clusters
ari_7 = adjusted_rand_score(ground_truth_labels, cluster_labels_7)
nmi_7 = normalized_mutual_info_score(ground_truth_labels, cluster_labels_7)

# Print results
print("K-means with 5 clusters (optimal from elbow method):")
print(f"Adjusted Rand Index (ARI): {ari_5:.4f}")
print(f"Normalised Mutual Information (NMI): {nmi_5:.4f}")
print("\nK-means with 7 clusters (ground truth):")
print(f"Adjusted Rand Index (ARI): {ari_7:.4f}")
print(f"Normalised Mutual Information (NMI): {nmi_7:.4f}")
```

```
K-means with 5 clusters (optimal from elbow method):
Adjusted Rand Index (ARI): 0.0578
Normalised Mutual Information (NMI): 0.1449

K-means with 7 clusters (ground truth):
Adjusted Rand Index (ARI): 0.0739
Normalised Mutual Information (NMI): 0.1640
```

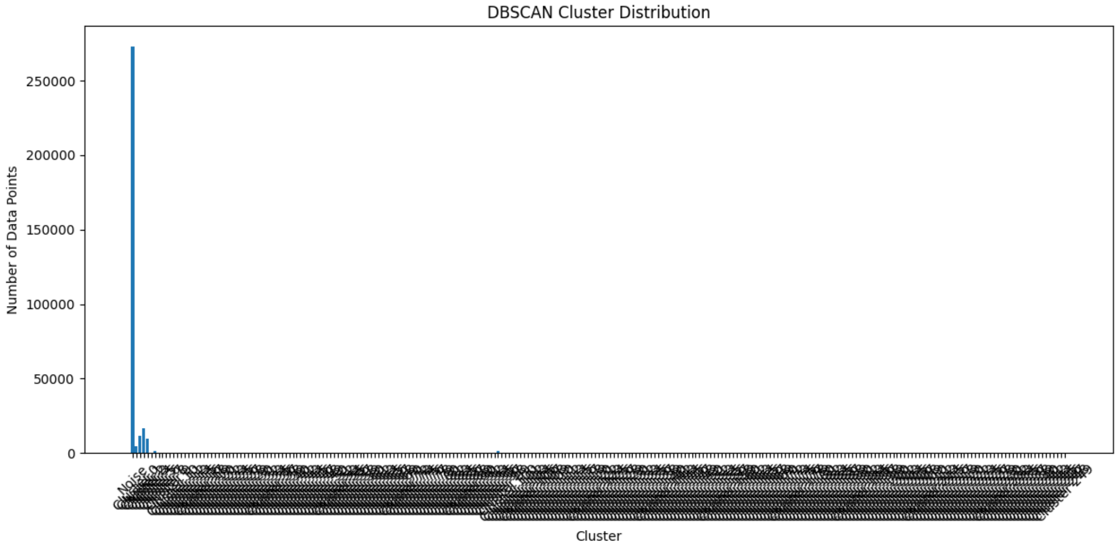
[ ]:
```python
# Implementing DBSCAN for cluster discovery
from sklearn.cluster import DBSCAN
from sklearn.neighbors import NearestNeighbors
import numpy as np
import matplotlib.pyplot as plt
from collections import Counter

# Function to find optimal epsilon using k-distance graph
def find_optimal_eps(data, k=5):
    # Calculate distances to k nearest neighbors for each point
    neigh = NearestNeighbors(n_neighbors=k)
    neigh.fit(data)
    distances, _ = neigh.kneighbors(data)

    # Sort distances to kth neighbor in ascending order
    k_distances = np.sort(distances[:, k-1])

    # Plot k-distance graph
    plt.figure(figsize=(12, 6))
    plt.plot(range(len(k_distances)), k_distances)
    plt.xlabel('Data Points (sorted by distance)')
    plt.ylabel(f'Distance to {k}th Nearest Neighbor')
    plt.title('K-Distance Graph for DBSCAN Epsilon Parameter Selection')
```

```
    # Add a grid to help identify the "elbow"
    plt.grid(True)
    plt.show()

    return k_distances

# Find optimal epsilon value
k_distances = find_optimal_eps(df_clean[numeric_cols])

# Based on the k-distance graph, we can identify the "elbow" point
# Let's try a range of epsilon values and min_samples
# eps_values = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7]
# min_samples_values = [5, 10, 15]
eps_values = [0.1, 0.5, 0.7]
min_samples_values = [15]


results = []

for eps in eps_values:
    for min_samples in min_samples_values:
        # Apply DBSCAN
        dbscan = DBSCAN(eps=eps, min_samples=min_samples)
        cluster_labels = dbscan.fit_predict(df_clean[numeric_cols])

        # Count number of clusters (excluding noise points labeled as -1)
        n_clusters = len(set(cluster_labels)) - (1 if -1 in cluster_labels else␣
↪0)
        noise_points = list(cluster_labels).count(-1)

        # Calculate evaluation metrics
        if n_clusters > 0:  # Only calculate metrics if clusters were found
            ari = adjusted_rand_score(ground_truth_labels, cluster_labels)
            nmi = normalized_mutual_info_score(ground_truth_labels,␣
↪cluster_labels)
        else:
            ari = 0
            nmi = 0

        # Store results
        results.append({
            'eps': eps,
            'min_samples': min_samples,
            'n_clusters': n_clusters,
            'noise_points': noise_points,
            'noise_percentage': noise_points / len(cluster_labels) * 100,
```

```python
            'ari': ari,
            'nmi': nmi
        })

# Convert results to DataFrame for easier analysis
import pandas as pd
results_df = pd.DataFrame(results)

# Display results sorted by ARI (higher is better)
print("DBSCAN Results sorted by ARI:")
print(results_df.sort_values('ari', ascending=False).head(10))

# Select the best parameter combination based on ARI
best_params = results_df.loc[results_df['ari'].idxmax()]
print("\nBest DBSCAN Parameters:")
print(f"Epsilon: {best_params['eps']}")
print(f"Min Samples: {best_params['min_samples']}")
print(f"Number of Clusters: {best_params['n_clusters']}")
print(f"Adjusted Rand Index: {best_params['ari']:.4f}")
print(f"Normalised Mutual Information: {best_params['nmi']:.4f}")
print(f"Noise Points: {best_params['noise_points']} ({best_params['noise_percentage']:.2f}%)")

# Apply DBSCAN with the best parameters
best_dbscan = DBSCAN(eps=best_params['eps'], min_samples=int(best_params['min_samples']))
best_cluster_labels = best_dbscan.fit_predict(df_clean[numeric_cols])

# Visualise cluster distribution
plt.figure(figsize=(12, 6))
cluster_counts = Counter(best_cluster_labels)
labels = [f"Cluster {i}" if i >= 0 else "Noise" for i in sorted(cluster_counts.keys())]
counts = [cluster_counts[i] for i in sorted(cluster_counts.keys())]

plt.bar(labels, counts)
plt.xlabel('Cluster')
plt.ylabel('Number of Data Points')
plt.title('DBSCAN Cluster Distribution')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()

# Compare with ground truth
print("\nComparison with Ground Truth:")
print(f"DBSCAN found {best_params['n_clusters']} clusters (excluding noise)")
print(f"Ground truth has {unique_locations} unique sensor locations")
```




```
DBSCAN Results sorted by ARI:
   eps  min_samples  n_clusters  noise_points  noise_percentage       ari  \
0  0.3           10         250        273067          84.836086  0.020353

        nmi
0  0.162161

Best DBSCAN Parameters:
Epsilon: 0.3
Min Samples: 10.0
Number of Clusters: 250.0
Adjusted Rand Index: 0.0204
Normalised Mutual Information: 0.1622
Noise Points: 273067.0 (84.84%)
```




```
Comparison with Ground Truth:
DBSCAN found 250.0 clusters (excluding noise)
Ground truth has 11 unique sensor locations
```