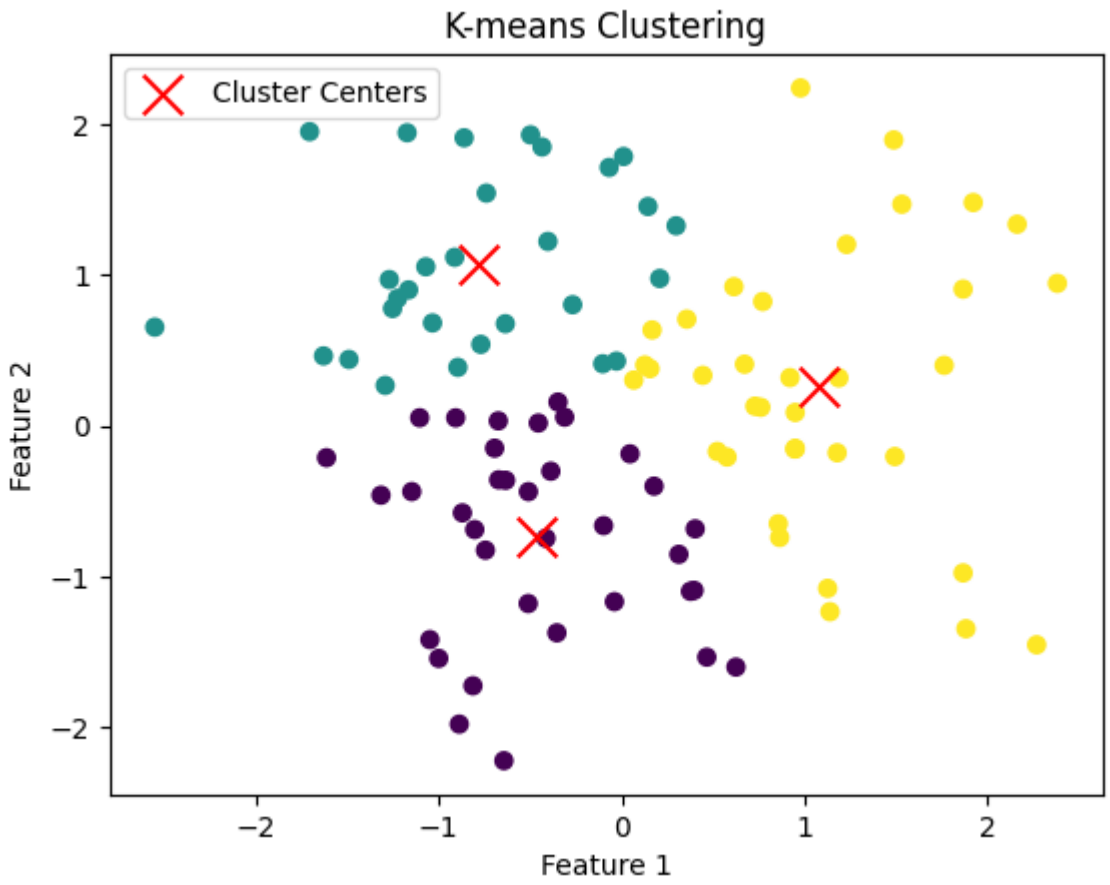
Clustering

In [14]:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans

# Generate some random data for clustering
np.random.seed(0)
data = np.random.randn(100, 2)  # 100 data points in 2 dimensions

# Create a K-means model with 3 clusters
kmeans = KMeans(n_clusters=3, random_state=0)

# Fit the model to the data
kmeans.fit(data)

# Get cluster assignments for each data point
cluster_labels = kmeans.labels_

# Get the coordinates of the cluster centers
cluster_centers = kmeans.cluster_centers_

# Visualize the data and clusters
plt.scatter(data[:, 0], data[:, 1], c=cluster_labels, cmap='viridis')
plt.scatter(cluster_centers[:, 0], cluster_centers[:, 1], c='red', marker='
plt.title("K-means Clustering")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.legend()
plt.show()
```

```
C:\Users\Rubab\AppData\Local\Programs\Python\Python310\lib\site-packages
\sklearn\cluster\_kmeans.py:1412: FutureWarning: The default value of `n_
init` will change from 10 to 'auto' in 1.4. Set the value of `n_init` exp
licitly to suppress the warning
  super()._check_params_vs_input(X, default_n_init=10)
```



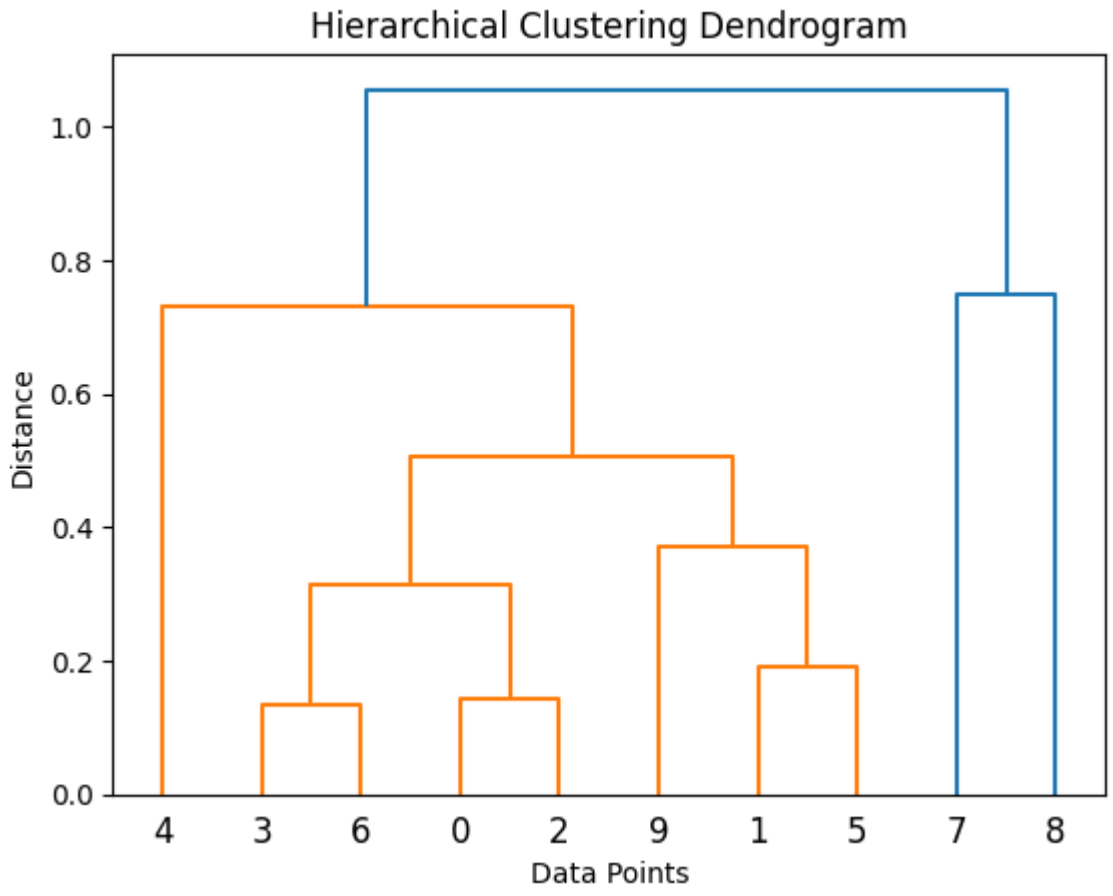
In [ ]:

```
# Create a KMeans model with 3 clusters: model
model = KMeans(n_clusters=3)

# Use fit_predict to fit model and obtain cluster labels: labels
labels = model.fit_predict(samples)

# Create a DataFrame with labels and varieties as columns: df
df = pd.DataFrame({'labels': labels, 'varieties': varieties})

# Create crosstab: ct
ct = pd.crosstab(df['labels'], df['varieties'])

# Display ct
print(ct)
```

Visualization with hierarchical clustering and t-SNE

In [2]:

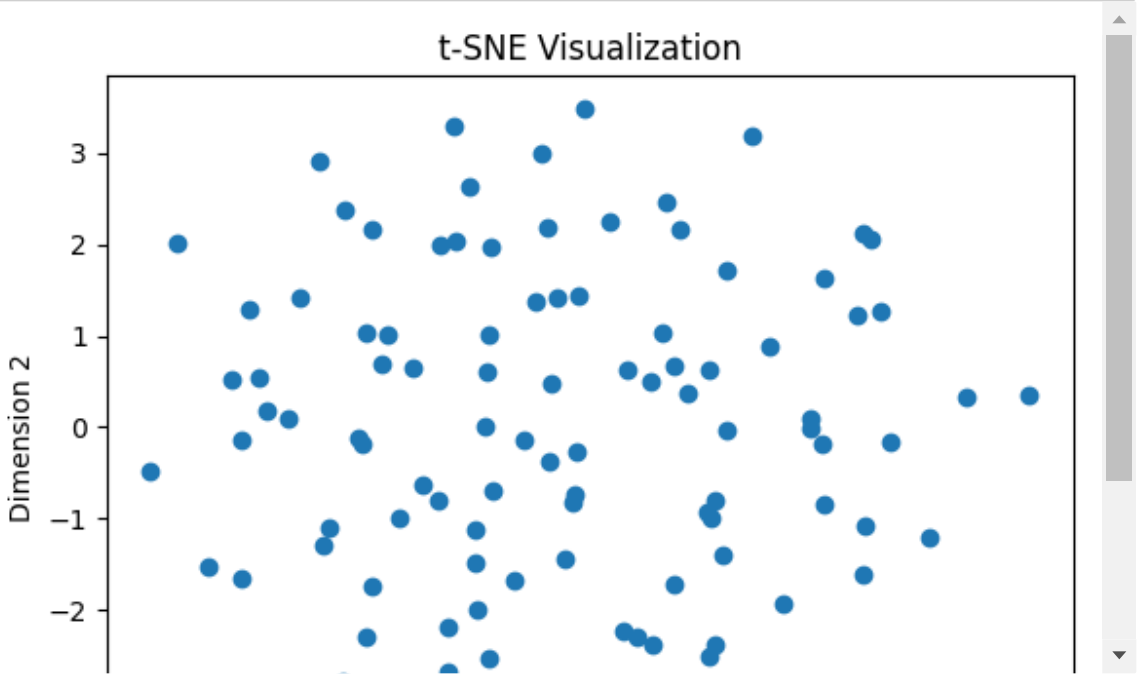
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.cluster.hierarchy import linkage, dendrogram

# Generate random data for clustering
np.random.seed(0)
data = np.random.rand(10, 2)

# Perform hierarchical clustering using the "ward" method
linkage_result = linkage(data, method='complete')

# Visualize the dendrogram
dendrogram(linkage_result)

plt.title('Hierarchical Clustering Dendrogram')
plt.xlabel('Data Points')
plt.ylabel('Distance')
plt.show()
```



TSNE

In [13]:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.manifold import TSNE

# Generate some random high-dimensional data
np.random.seed(0)
data = np.random.randn(100, 50)  # 100 data points in 50 dimensions

# Create a t-SNE model with two output dimensions
tsne = TSNE(n_components=2, perplexity=30, random_state=0)

# Fit the model to the data and transform the data into the lower-dimensior
tsne_result = tsne.fit_transform(data)

# Visualize the t-SNE results
plt.scatter(tsne_result[:, 0], tsne_result[:, 1])
plt.title("t-SNE Visualization")
plt.xlabel("Dimension 1")
plt.ylabel("Dimension 2")
plt.show()
```



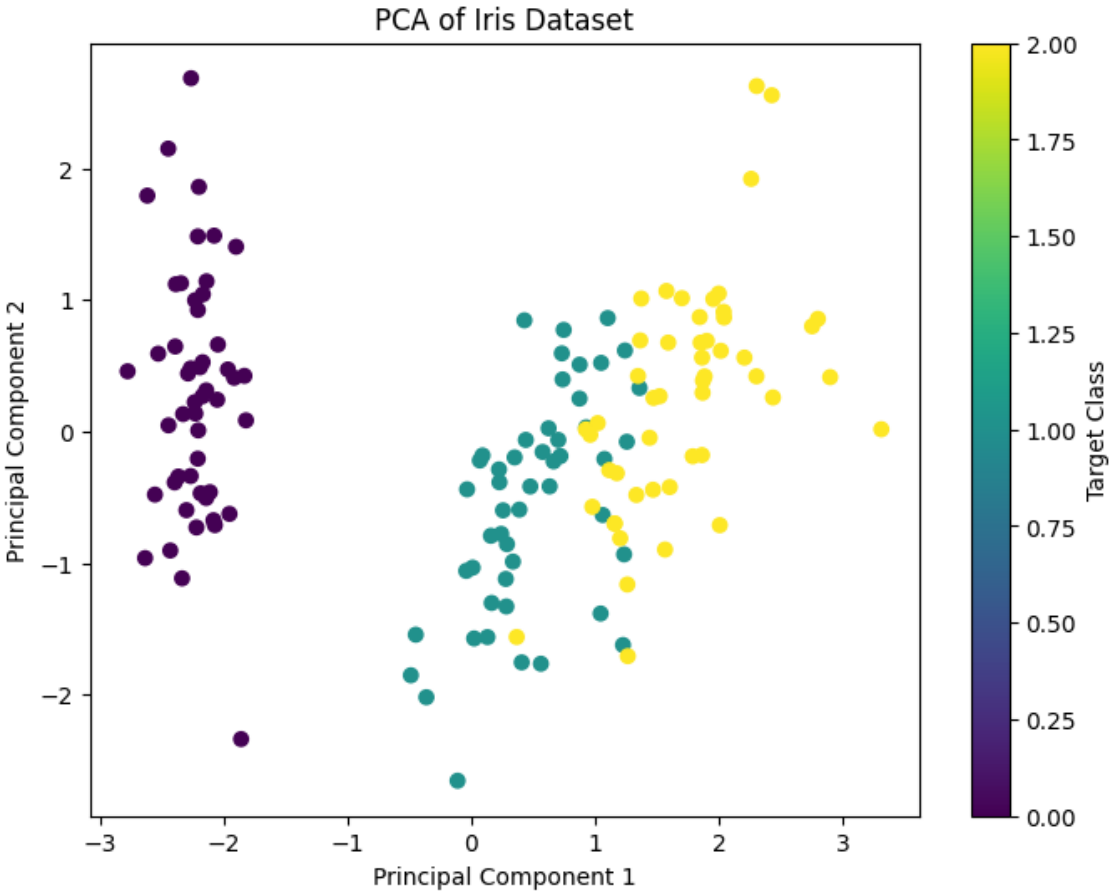
Decorrelating Your Data and Dimension Reduction

Dimension reduction summarizes a dataset using its common occuring patterns.
Principal Component Analysis (PCA) is a method to simplify and understand data.

In [1]:

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.datasets import load_iris

# Load the Iris dataset as an example
iris = load_iris()
X = iris.data  # Feature matrix

# Standardize the data (mean = 0, variance = 1)
mean = np.mean(X, axis=0)
std_dev = np.std(X, axis=0)
X_standardized = (X - mean) / std_dev

# Create a PCA model with two principal components
pca = PCA(n_components=2)

# Fit the PCA model to the standardized data
principal_components = pca.fit_transform(X_standardized)

# Explained variance ratio
explained_variance_ratio = pca.explained_variance_ratio_
print("Explained Variance Ratio:", explained_variance_ratio)

# Plot the data in the reduced principal component space
plt.figure(figsize=(8, 6))
plt.scatter(principal_components[:, 0], principal_components[:, 1], c=iris.
plt.title("PCA of Iris Dataset")
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.colorbar(label="Target Class")
plt.show()
```

```
Explained Variance Ratio: [0.72962445 0.22850762]
```



Non-negative matrix factorization ("NMF")

(NMF) is a dimensionality reduction technique that factors a matrix into two lower-dimensional matrices, where all values are non-negative.

In [7]:

```python
import numpy as np
from sklearn.decomposition import NMF

# Create a random matrix
X = np.random.rand(10, 5)

# Create an NMF model with 2 components
model = NMF(n_components=2, init='random', random_state=0)

# Fit the model to the data
model.fit(X)

# Get the fitted components
W = model.components_
H = model.transform(X)

# Print the components
print(X,'\n')
print(W,'\n')
print(H)
```

```
[[0.19007261 0.842583   0.40599708 0.33239028 0.24861076]
 [0.4055105  0.28584445 0.85291228 0.15983596 0.24175775]
 [0.13402136 0.14398828 0.97597876 0.96570981 0.86817148]
 [0.65540136 0.10596605 0.6929895  0.02619954 0.2839503 ]
 [0.39417747 0.16997789 0.17721685 0.9131456  0.23919291]
 [0.10850101 0.65924644 0.56868091 0.4172795  0.29075507]
 [0.33487275 0.91859539 0.17862016 0.97102731 0.50206652]
 [0.34275763 0.83468802 0.67211062 0.62142735 0.8164877 ]
 [0.91339184 0.92221645 0.5377335  0.38237542 0.24757554]
 [0.59096319 0.52237148 0.94579176 0.07966467 0.49469438]]

[[0.22150886 0.93535171 0.31716396 1.37098387 0.71961114]
 [0.48640557 0.32396829 0.74172394 0.         0.28264516]]

[[0.35435245 0.44858328]
 [0.05772384 0.96931261]
 [0.55143273 0.63065288]
 [0.         0.9750883 ]
 [0.49890625 0.02271973]
 [0.34668791 0.48977006]
 [0.761421   0.09069122]
 [0.5559643  0.71448639]
 [0.32785423 0.96068538]
 [0.09329679 1.25441139]]
```