```python
import numpy as np
import scipy as sp


def k_init(X, k):
    """ k-means++: initialization algorithm

    Parameters
    ----------
    X: array, shape(n ,d)
        Input array of n samples and d features

    k: int
        The number of clusters

    Returns
    -------
    init_centers: array (k, d)
        The initialize centers for kmeans++
    """
    n_samples, d_features = X.shape
    centers = [X[np.random.randint(0,n_samples)]] #randomly choose a first center for the
clusters

    for i in range(1,k):
        distances = np.array([min(np.sum((x - c) ** 2) for c in centers) for x in X])
        probabilities = distances/np.sum(distances)
        cum_probabilities = np.cumsum(probabilities)
        r = np.random.rand()
        for index, prob in enumerate(cum_probabilities):
            if r < prob:
                centers.append(X[index])
                break

    return np.array(centers)

def k_means_pp(X, k, max_iter):
    """ k-means++ clustering algorithm

    step 1: call k_init() to initialize the centers
    step 2: iteratively refine the assignments

    Parameters
    ----------
    X: array, shape(n ,d)
        Input array of n samples and d features

    k: int
        The number of clusters

    max_iter: int
        Maximum number of iteration

    Returns
    -------
    final_centers: array, shape (k, d)
        The final cluster centers
    """
    centers = k_init(X, k)

    for i in range(max_iter):
        data_map = assign_data2clusters(X, centers)
        new_centers = np.zeros_like(centers)

        for j in range(k):
            assigned_points = X[data_map[:,j]==1]
            if len(assigned_points) > 0:
```

```
                new_centers[j] = np.mean(assigned_points, axis = 0)
            else:
                new_centers[j] = X[np.random.randint(0, X.shape[0])]

        if np.allclose(centers, new_centers):
            break
        centers = new_centers

    return centers


def assign_data2clusters(X, C):
    """ Assignments of data to the clusters
    Parameters
    ----------
    X: array, shape(n ,d)
        Input array of n samples and d features

    C: array, shape(k ,d)
        The final cluster centers

    Returns
    -------
    data_map: array, shape(n, k)
        The binary matrix A which shows the assignments of data points (X) to
        the input centers (C).
    """
    n = X.shape[0]
    k = C.shape[0]
    data_map = np.zeros((n, k))

    for i in range(n):
        distances = np.linalg.norm(X[i] - C, axis = 1)
        cluster = np.argmin(distances)
        data_map[i, cluster] = 1

    return data_map


def compute_objective(X, C):
    """ Compute the clustering objective for X and C
    Parameters
    ----------
    X: array, shape(n ,d)
        Input array of n samples and d features

    C: array, shape(k ,d)
        The final cluster centers

    Returns
    -------
    accuracy: float
        The objective for the given assigments
    """
    n = X.shape[0]
    total_distortion = 0.0

    for i in range(n):
        distances = np.linalg.norm(X[i] - C, axis = 1)
        min_dist = np.min(distances)
        total_distortion += min_dist ** 2

    return total_distortion
```

Homework 5

Problem 1

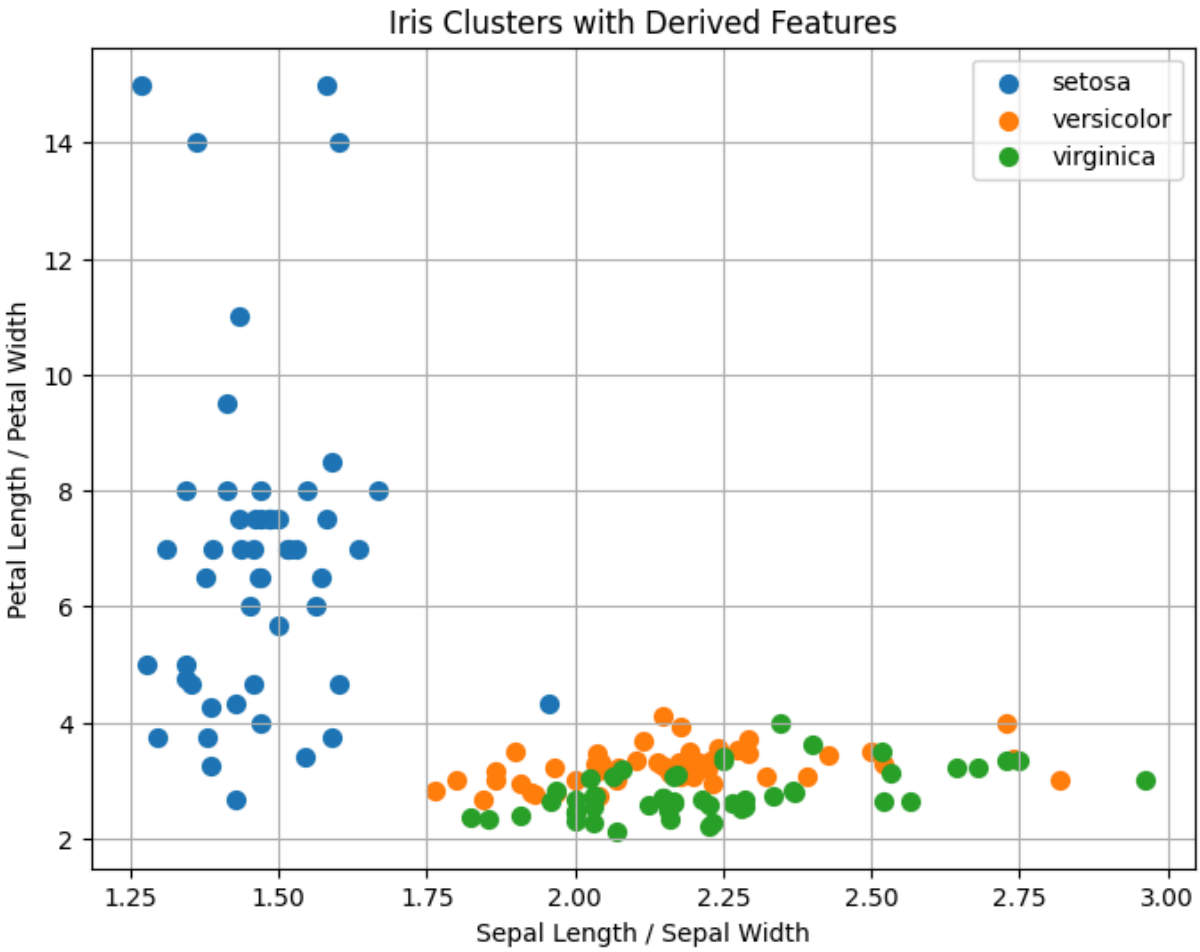
Load Data

In [1]:
```
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

iris = load_iris()
X = iris.data
y = iris.target
```

a)

In [2]:
```
iris_df = pd.DataFrame(X, columns=iris.feature_names)

df = pd.DataFrame()
df['x1'] = iris_df['sepal length (cm)'] / iris_df['sepal width (cm)']
df['x2'] = iris_df['petal length (cm)'] / iris_df['petal width (cm)']

df['target'] = y
df['target_name'] = df['target'].apply(lambda i: iris.target_names[i])
```

In [3]:
```
plt.figure(figsize=(8, 6))
for name, group in df.groupby('target_name'):
    plt.scatter(group['x1'], group['x2'], label=name, s=50)

plt.xlabel('Sepal Length / Sepal Width')
plt.ylabel('Petal Length / Petal Width')
plt.title('Iris Clusters with Derived Features')
plt.legend()
plt.grid(True)
plt.show()
```



b)

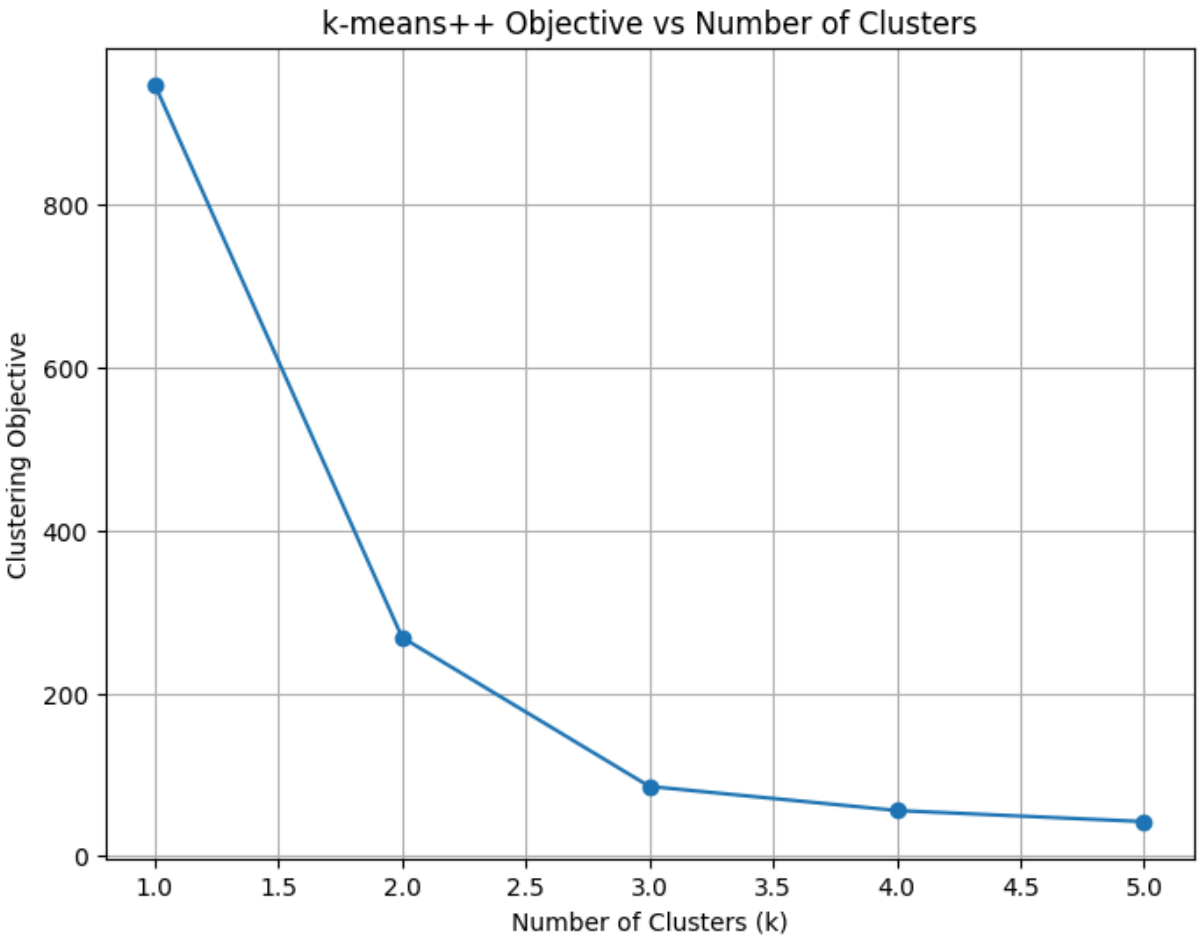
Done in the Problem1.py file

In [10]:
```
from Problem1 import *
```

c)

In [5]:
```
results = {}

for k in range(1, 6):
    best_obj = float('inf')
    for _ in range(50):
        centers = k_means_pp(df[['x1', 'x2']].values, k, max_iter=100)
        obj = compute_objective(df[['x1', 'x2']].values, centers)
        if obj < best_obj:
            best_obj = obj
    results[k] = best_obj

plt.figure(figsize=(8, 6))
plt.plot(list(results.keys()), list(results.values()), marker='o')
plt.xlabel('Number of Clusters (k)')
plt.ylabel('Clustering Objective')
plt.title('k-means++ Objective vs Number of Clusters')
```

```
plt.grid(True)
plt.show()
```



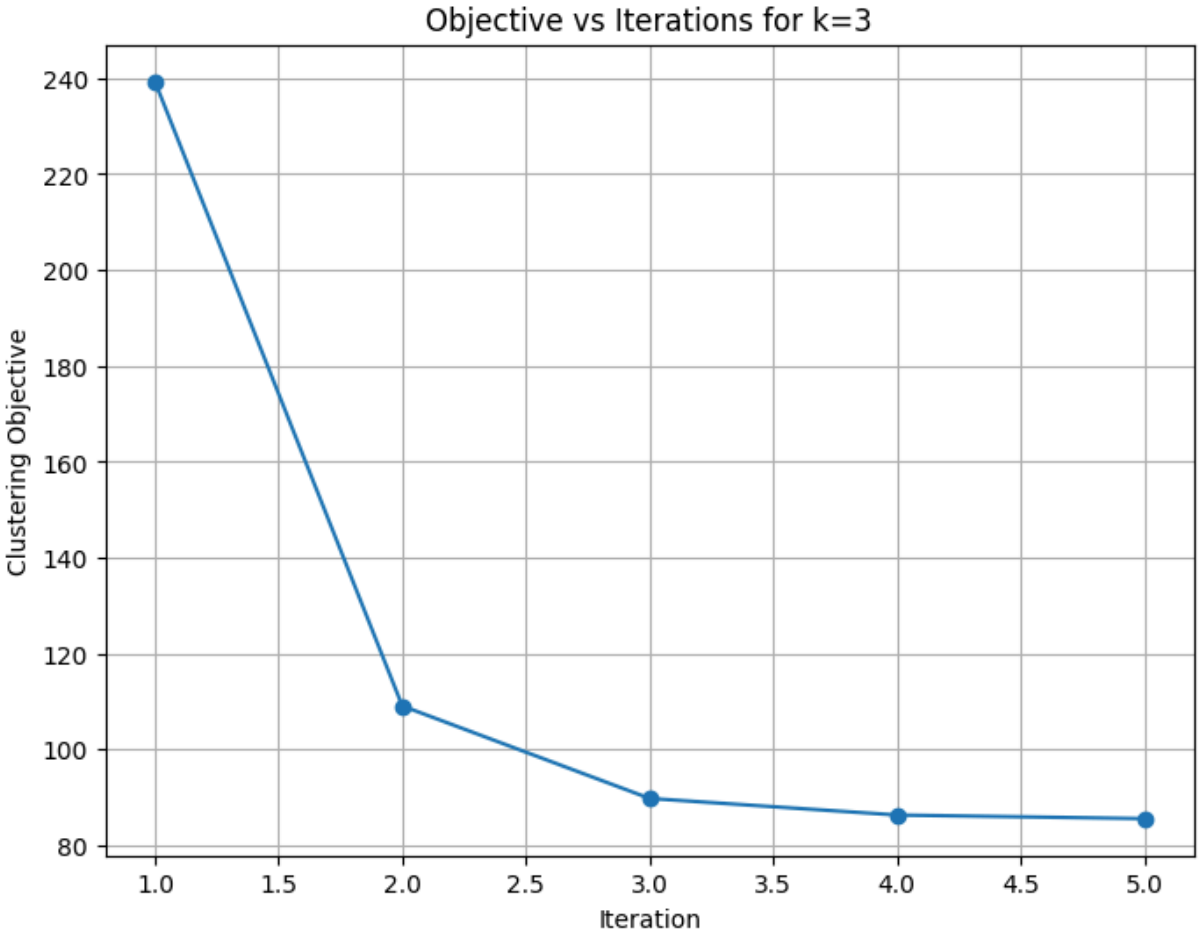
d) Based on the above plot, decide the number of final clusters and justify your answer.

In this scenario is when we have to use the 'elbow' method that was discussed in class. Here we will look for where the graph of the objective starts to stop changing so drastically. We can see here that the graph derivative starts to flatten at about k=3. So we will determine that the best number of clusters for this data set is 3. This also makes sense with our prior knowledge as the iris dataset has 3 classes.

e)

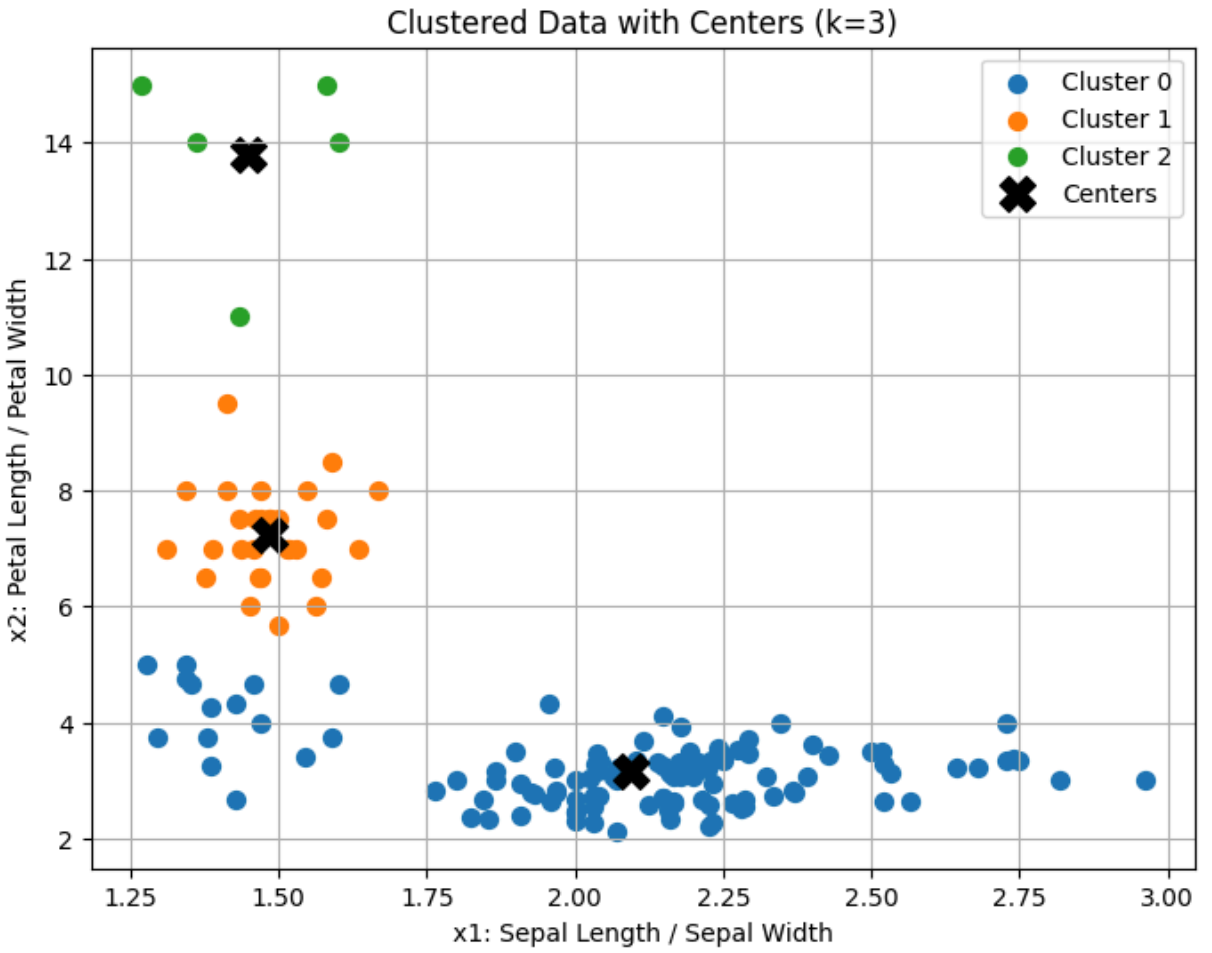
In [13]:
```
X_mod = df[['x1', 'x2']].values
k = 3

objectives = []
centers = k_init(X_mod, k)
for _ in range(100):
    data_map = assign_data2clusters(X_mod, centers)
    new_centers = np.zeros_like(centers)
    for j in range(k):
```

```
        assigned_points = X_mod[data_map[:, j] == 1]
        if len(assigned_points) > 0:
            new_centers[j] = np.mean(assigned_points, axis=0)
        else:
            new_centers[j] = X_mod[np.random.randint(0, X_mod.shape[0])]
    obj = compute_objective(X_mod, centers)
    objectives.append(obj)
    if np.allclose(centers, new_centers):
        break
    centers = new_centers

plt.figure(figsize=(8, 6))
plt.plot(range(1, len(objectives) + 1), objectives, marker='o')
plt.xlabel('Iteration')
plt.ylabel('Clustering Objective')
plt.title('Objective vs Iterations for k=3')
plt.grid(True)
plt.show()
```



In [12]:
```
final_assignments = assign_data2clusters(X_mod, centers)

cluster_ids = np.argmax(final_assignments, axis=1)

plt.figure(figsize=(8, 6))
for cluster_id in range(k):
    cluster_points = X_mod[cluster_ids == cluster_id]
    plt.scatter(cluster_points[:, 0], cluster_points[:, 1], label=f'Cluster {cluste
```

```
plt.scatter(centers[:, 0], centers[:, 1], c='black', marker='X', s=200, label='Cent
plt.xlabel('x1: Sepal Length / Sepal Width')
plt.ylabel('x2: Petal Length / Petal Width')
plt.title('Clustered Data with Centers (k=3)')
plt.legend()
plt.grid(True)
plt.show()
```



In [ ]:

Problem 2

$X \in \mathbb{R}^{n \times d}$

a) centering

meaning: subtract the mean of each feature from the data, so each feature has a mean of 0

importance: centering makes sure that the first PC captures the direction of highest variance

application: $\mu_j = \frac{1}{n}\sum_{i=1}^{n} X_{ij}$ centered matrix: $\hat{X}$ given by $\hat{X}_{ij} = X_{ij} - \mu_j$ for all $i = 1, \dots n$ and $j = 1 \dots d$

b) scaling

meaning: rescales a feature so each value falls between range a to b

importance: when features have different scales, it may dominate the other features scaling makes sure that all features contribute equally

application: to apply use the min/max of each feature

X_j^{min} = smallest i in X_j column X_j^{max} = largest i in X_j column

so $\hat{X}_{ij} = a + \frac{(x_{ij} - X_j^{min})(b-a)}{X_j^{max} - X_j^{min}}$

C) Standardization

meaning: standardizes features so each has mean=0 and Var = 1

importance: makes sure that all features are on the same scale, otherwise features with large variances will dominate

application: μ_j = mean of feature j σ_j = sd of feature j z-score

$$\hat{X}_{ij} = \frac{X_{ij} - \mu_j}{\sigma_j}$$

d) normalization

meaning: normalizes each data point so the vector has length 1

importance: used when the direction of the data matters more than the length. Takes importance away from magnitude

application: divide each row of data by its length $\hat{X}_{i:} = \frac{X_{i:}}{\|X_{i:}\|_2}$

B) $X = \begin{bmatrix} 1 & 2 \\ -1 & 1 \\ 0 & 1 \\ 2 & 4 \\ 3 & 1 \end{bmatrix}$ $n=5$ $d=2$

Centering: $\mu_1 = \frac{1}{5}(1-1+0+2+3) = 1$ $\mu_2 = \frac{1}{5}(2+1+1+4+1) = \frac{9}{5}$

$\hat{X} = \begin{bmatrix} 0 & 0.2 \\ -2 & -0.8 \\ -1 & -0.8 \\ 1 & 2.2 \\ 2 & -0.8 \end{bmatrix}$

Scaling to [0,1]: $X_1^{min} = -1$ $X_1^{max} = 3$ $X_2^{min} = 1$ $X_2^{max} = 4$ $\hat{X}_{ij} = \frac{X_{ij} - X_j^{min}}{X_j^{max} - X_j^{min}}$

$\hat{X} = \begin{bmatrix} 0.5 & 0.333 \\ 0 & 0 \\ 0.25 & 0 \\ 0.75 & 1 \\ 1 & 0 \end{bmatrix}$

Standardization: $\mu_1 = 1$ $\sigma_1 = \sqrt{2}$ $\mu_2 = 1.8$ $\sigma_2 = \sqrt{1.36}$

$\hat{X} = \begin{bmatrix} 0 & 0.171 \\ -\sqrt{2} & -0.686 \\ -1/\sqrt{2} & -0.686 \\ 1/\sqrt{2} & 1.886 \\ \sqrt{2} & -0.686 \end{bmatrix}$

Normalization:

row 1 = $\sqrt{1^2+2^2} = \sqrt{5}$

row 2 = $\sqrt{(-1)^2+1^2} = \sqrt{2}$

row 3 = 1

row 4 = $\sqrt{20}$

row 5 = $\sqrt{10}$

divide each row

$\hat{X} = \begin{bmatrix} 0.447 & 0.894 \\ -0.707 & 0.707 \\ 0 & 1 \\ 0.447 & 0.894 \\ 0.949 & 0.316 \end{bmatrix}$

Problem 4

$$\min_{U,V}\left[f(U,V) \equiv \sum_{(i,j)\in\Omega}(R_{ij} - u_i^T v_j)^2 + \alpha\sum_{i=1}^{n}\|u_i\|_2^2 + \beta\sum_{j=1}^{m}\|v_j\|_2^2\right]$$

1) If we set $U^{(0)}$ and $V^{(0)}$ to zero the algorithm will get stuck since $U^{(0)}$ and $V^{(0)}$ will all be 0s, the respective vectors will also be zero vectors $(U_i; V_j)$. Then in the objective $u_i^T v_j = 0$ meaning that the pretictions will be 0 for all i and j

since we ~~set~~ change one of u and v and fix the other in alternating method

$\nabla_U f = -2\sum(R_{ij} - u_i^T v_j)v_j + 2\alpha u_i$ if V is fixed to 0 the entire gradient will be 0 since U is initialized to 0. The update will then set U to 0 and this will continue. This same thing happens when trying to update V as well.

2) looking at the slides from class we get that the updates are:

$\hat{u}_i = \left(\sum v_j v_j^T + \underset{\beta}{\lambda} I\right)^{-1}\left(\sum r_{ij} v_j\right)$ $\quad \hat{v}_j = \left(\sum u_i u_i^T + \underset{\alpha}{\lambda} I\right)^{-1}\left(\sum r_{ij} u_i\right)$

if there is no regularization $\alpha = \beta = 0$

then the $v_j v_j^T$ and $u_i u_i^T$ will be a $k \times k$ matrix that is to be inverted

We need to ensure that it has full rank to be inverted so we need at least k movies and k users for it to be invertible.

3) objective to minimize is $f(U,V)$

Loss at iteration t: $(r - u^T v)^2 + \alpha\|u\|^2 + \beta\|v\|^2$

$\nabla_U L = -2(r - u^T v)v + 2\alpha u$ $\quad \nabla_V L = -2(r - u^T v) + 2\beta v$

```
def SGD(R, k, α, β, η, T):
    for each user i: u_i = random vector in R^k
    for each movie j: v_j = random vector in R^k
    for t = 1 to T:
        randomly sample user i and movie j that is observed record r_ij as well
        Then update the error to r_ij - u_i^T v_j
        Next update gradients
            u_i = u_i + η ∇_U f(U,V)
            v_j = v_j + η ∇_V f(U,V)
    return u_i and v_j
```

Problem 3

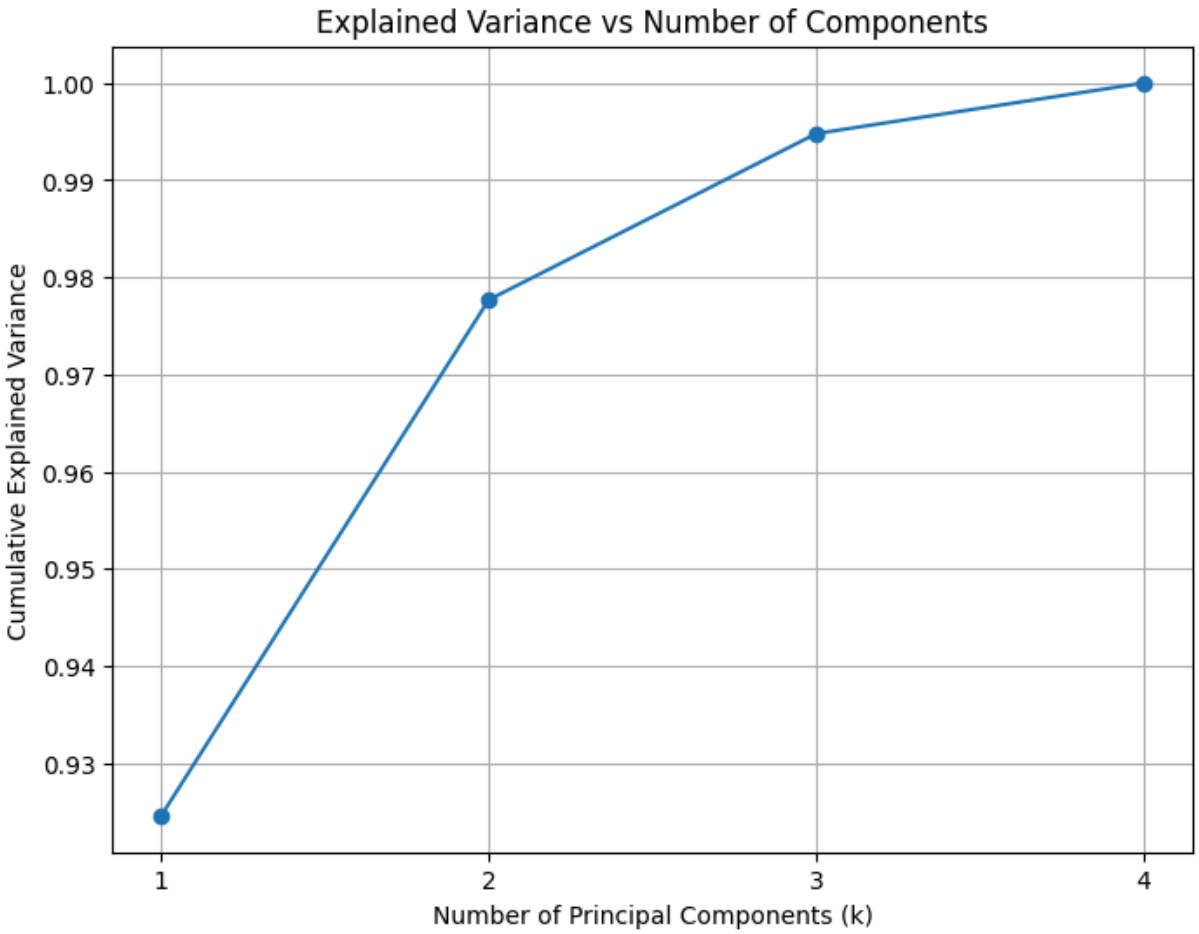
Load data

In [2]:
```
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
from sklearn.decomposition import PCA

iris = load_iris()
X = iris.data
y = iris.target
```

a)

In [4]:
```
explained_variances = []

for k in range(1, 5):
    pca = PCA(n_components=k)
    pca.fit(X)
    explained_variances.append(np.sum(pca.explained_variance_ratio_))

# Print variance explained by the first component
pca_1 = PCA(n_components=1)
pca_1.fit(X)
first_pc_var = pca_1.explained_variance_ratio_[0]
print(f"Variance explained by the first principal component: {first_pc_var:.4f}")

# Plot cumulative explained variance
plt.figure(figsize=(8, 6))
plt.plot(range(1, 5), explained_variances, marker='o')
plt.xlabel('Number of Principal Components (k)')
plt.ylabel('Cumulative Explained Variance')
plt.title('Explained Variance vs Number of Components')
plt.grid(True)
plt.xticks([1, 2, 3, 4])
plt.show()
```

```
Variance explained by the first principal component: 0.9246
```



According to the graph and the output of the code, 92.46% of the variance in the data is explained by the first principal component

As we keep adding components to the algorithm, we continue to increase the variance that is explained by the components. For example, as we increase to 2 PCAs, about 97.7% of the variance is explained by the first two PCAs. Then 99.5% is explained by the first three PCAs. And finally 100% of the variance is explained by four PCAs, this is because we are using all of the features in the dataset.

b)

In [7]:
```
from sklearn.preprocessing import MinMaxScaler, StandardScaler, Normalizer

methods = {
    "a) Centering": lambda X: X - X.mean(axis=0),
    "b) Scaling": lambda X: MinMaxScaler().fit_transform(X),
    "c) Standardization": lambda X: StandardScaler().fit_transform(X),
    "d) Normalization": lambda X: Normalizer().fit_transform(X)
}

X_raw = iris.data

for label, transform in methods.items():
    X_proc = transform(X_raw)
```

```
    explained = []
    for k in range(1, 5):
        pca = PCA(n_components=k)
        pca.fit(X_proc)
        explained.append(np.sum(pca.explained_variance_ratio_))

    print(f"\n{label}")
    for i, var in enumerate(explained, start=1):
        print(f"  Components: {i}, Explained Variance: {var:.4f}")
```

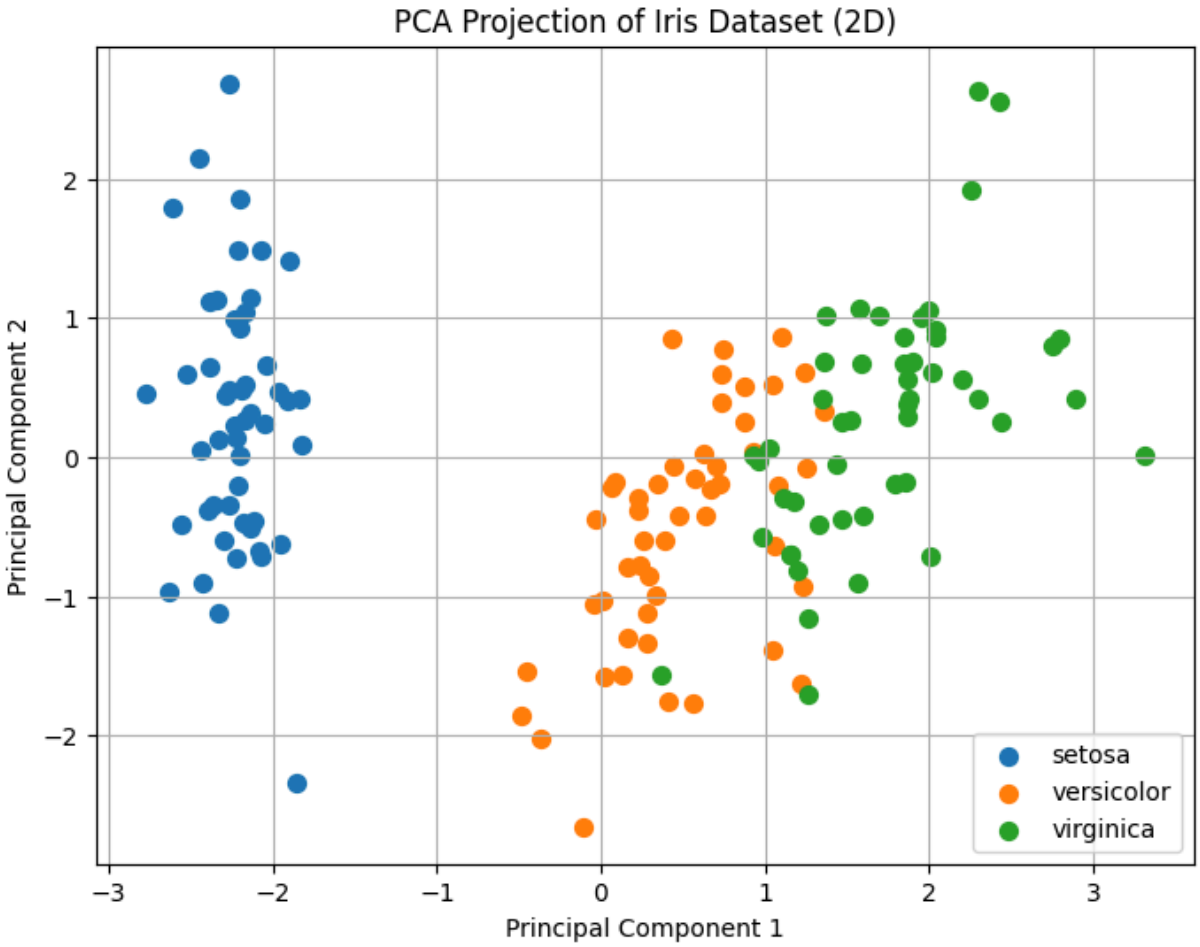
```
a) Centering
  Components: 1, Explained Variance: 0.9246
  Components: 2, Explained Variance: 0.9777
  Components: 3, Explained Variance: 0.9948
  Components: 4, Explained Variance: 1.0000

b) Scaling
  Components: 1, Explained Variance: 0.8414
  Components: 2, Explained Variance: 0.9589
  Components: 3, Explained Variance: 0.9936
  Components: 4, Explained Variance: 1.0000

c) Standardization
  Components: 1, Explained Variance: 0.7296
  Components: 2, Explained Variance: 0.9581
  Components: 3, Explained Variance: 0.9948
  Components: 4, Explained Variance: 1.0000

d) Normalization
  Components: 1, Explained Variance: 0.9624
  Components: 2, Explained Variance: 0.9900
  Components: 3, Explained Variance: 0.9981
  Components: 4, Explained Variance: 1.0000
```

According to the operations done in problem 2, we have an output that shows all of the variances for each component. When centering the data, we get exactly the same variances as when the data was uncentered. Scaling results in a lower variance for all of the components compared to centering or just the raw data. With standardization, PC1 explains even less variance (only ~73%) compared to other methods; this is because standardization equalizes feature variances, which can suppress dominant features — leading to a more balanced variance distribution. With Normalization, PC1 explains the most variance as compared to the other methods. This is because normalization makes all PCs have a unit norm, meaning that most variance should be concentrated in a single direction.

c)

In [9]:
```
X_std = StandardScaler().fit_transform(X)

pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_std)

df_pca = pd.DataFrame(X_pca, columns=['PC1', 'PC2'])
df_pca['target'] = y
```

```
df_pca['target_name'] = df_pca['target'].apply(lambda i: iris.target_names[i])

plt.figure(figsize=(8, 6))
for name, group in df_pca.groupby('target_name'):
    plt.scatter(group['PC1'], group['PC2'], label=name, s=50)

plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.title('PCA Projection of Iris Dataset (2D)')
plt.legend()
plt.grid(True)
plt.show()
```



Here, the different classes show some distinct seperation. We see that setosa forms a distinct group on the left of the plot, while versicolor and virginica are slightly seperable on the right side of the plot with just a little bit of overlap.

d)

In [10]:
```
from sklearn.cluster import KMeans

# Cluster the 2D PCA data
kmeans = KMeans(n_clusters=3, init='k-means++', random_state=448)
clusters = kmeans.fit_predict(X_pca)
```

```
# Add cluster labels to the DataFrame
df_pca['cluster'] = clusters

# Plot clusters
plt.figure(figsize=(8, 6))
for c in range(3):
    cluster_points = df_pca[df_pca['cluster'] == c]
    plt.scatter(cluster_points['PC1'], cluster_points['PC2'], label=f'Cluster {c}',

# Plot cluster centers
centers = kmeans.cluster_centers_
plt.scatter(centers[:, 0], centers[:, 1], c='black', marker='X', s=200, label='Cent

plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.title('K-Means++ Clustering on PCA-Reduced Iris Data (k=3)')
plt.legend()
plt.grid(True)
plt.show()
```

Here we see that the clusters found by using k++ means are slightly different than the groups found just by using the first 2 PCs. The cluster for setosa is correct, but it is the other two classes where the algorithm is slightly different.

In [ ]: