In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

In [20]:
```
df = pd.read_csv('Q1_Data\masked_data.csv')
df.head()
df.isna().sum().sum()
```

Out[20]:
```
2054
```

1a) PCA Implementation

In [45]:
```
df = df.fillna(df.mean())
df.isna().sum().sum()
```

Out[45]:
```
0
```

In [48]:
```
X = df.to_numpy()

def pca(X, rank):
    U, S, Vt = np.linalg.svd(X, full_matrices=False)

    U_d = U[:, :rank]  #truncate SVD
    S_d = np.diag(S[:rank])
    V_d = Vt[:rank, :]
    X_approx = np.dot(U_d, np.dot(S_d, V_d))

    return X_approx

rank = 2  # Perform PCA imputation with rank = 2
X_imputed = pca(X, rank)

print("Original Dataset (with mean imputation for missing values):")
print(X)
print("\nImputed Dataset after PCA reconstruction:")
print(X_imputed)
```

```
Original Dataset (with mean imputation for missing values):
[[-0.49502456 -1.82878363  0.4841028  ... -0.12573805  0.08584068
   0.34145413]
 [ 1.57260501  5.60961097  1.25805029 ... -0.12573805 -1.2757793
  -1.00174493]
 [ 2.61690054  4.26199367  0.30100245 ... -0.12573805  0.11701166
  -0.16071368]
 ...
 [ 1.91459015  1.49742945  1.42286307 ... -0.43355027 -0.05923609
  -0.51840571]
 [-0.56798275 -3.31736756  0.01489833 ... -0.50692899  0.11701166
   1.81593288]
 [-0.7891745  -3.61154383  0.10443142 ... -1.09058503  1.72809394
  -0.16071368]]

Imputed Dataset after PCA reconstruction:
[[-0.09680324 -0.91506569 -0.22616656 ...  0.49527196 -0.16571676
   0.55857537]
 [ 1.14922257  4.50046219  0.9457927  ... -1.74417047 -0.12607433
  -1.56102288]
 [ 1.33348612  2.32523888  0.30564732 ... -0.14105704 -1.09934465
   0.49697584]
 ...
 [ 0.91786277  0.63815245 -0.05265854 ...  0.52838298 -1.07331376
   1.10891464]
 [-0.11410553 -1.49876223 -0.38142856 ...  0.85686331 -0.33356319
   0.99319678]
 [-0.98032751 -2.45670879 -0.42895615 ...  0.58939201  0.56233882
   0.23010493]]
```

In [ ]:
```
```

In [47]:
```
def pca1(D, K):
    X = D.T
    U, S, Vt = np.linalg.svd(X, full_matrices=False)

    U_k = U[:, :K]   # truncate matrices for dimensionality reduction  # First K columns of U
    S_k = np.diag(S[:K])  # top k rank singular values

    Z = np.dot(U_k, S_k)

    return Z.T
```

```
K = 2
D_dimred = pca1(X, K)

print("Original Dataset:")
print(X)
print("\nReduced Dataset:")
print(D_dimred)
```

```
Original Dataset:
[[-0.49502456 -1.82878363  0.4841028  ... -0.12573805  0.08584068
   0.34145413]
 [ 1.57260501  5.60961097  1.25805029 ... -0.12573805 -1.2757793
  -1.00174493]
 [ 2.61690054  4.26199367  0.30100245 ... -0.12573805  0.11701166
  -0.16071368]
 ...
 [ 1.91459015  1.49742945  1.42286307 ... -0.43355027 -0.05923609
  -0.51840571]
 [-0.56798275 -3.31736756  0.01489833 ... -0.50692899  0.11701166
   1.81593288]
 [-0.7891745  -3.61154383  0.10443142 ... -1.09058503  1.72809394
  -0.16071368]]

Reduced Dataset:
[[-24.66251834 -68.68130365 -12.67108825  -0.65664502 -50.20435799
   -5.40622783   8.34002291 -22.75136997  35.57964059  39.30069824
   21.75047626  48.0825486   -4.58082184  17.21258743  39.04239538
  -15.6840096    6.35983856  19.29714372  11.88451313  11.26855804]
 [  8.59380283  -1.92637495  -2.65270708  15.02061166 -38.39381566
   21.04327323  10.27832767  -9.60851331 -32.87655979 -23.28980491
   24.80554415   1.63453466  16.58552136  11.35134999 -37.00189669
  -39.66330574 -19.54862517  10.08256742 -12.64880056  16.67863176]]
```

b) PPCA using Coordinate descent

In [68]:
```
def ppca(X, dim, epsilon=1e-4, max_iter=100):

    N, M = X.shape
    Z = np.zeros((N, dim))  # Latent variables
    mu = np.zeros(M)  # data mean
    W = np.zeros((M, dim))  #matrix
    sigma2 = 1.0  # Variance

    for iteration in range(max_iter):
        sigma2_old = sigma2
        Z_old = Z.copy()

        for n in range(N):  # E-step: Update latent variables Z_n for each data point
            xn = X[n, :]
            Z[n, :] = np.linalg.inv(W.T @ W + sigma2 * np.eye(dim)) @ (W.T @ (xn - mu))

        # M-step: Update parameters
        mu_old = mu.copy()  # Update mean (mu)
        mu = np.mean(X - (W @ Z.T).T, axis=0)

        residual = X - mu_old - (W @ Z.T).T  # Update variance (sigma^2)
        sigma2 = np.sum(residual ** 2) / (N)

        numerator = np.sum([(X[n, :] - mu_old)[:, None] @ Z[n, :][None, :] for n in range(N)], axis=0)
        denominator = np.sum([Z[n, :][:, None] @ Z[n, :][None, :] for n in range(N)], axis=0)
        cond_num = np.linalg.cond(denominator)  # Check for condition number to detect singularity
        if cond_num > 1e12:
            print("warning: matrix is singular")
            denominator += 1e-6 * np.eye(dim)
        W = np.linalg.inv(denominator) @ numerator  # Update W by getting the inverse of the denominator and so

        if np.mean(np.linalg.norm(Z - Z_old, axis=1)) < epsilon:
            break
    return Z  # Return the low-dimensional representation

dim = 2
X_imputed_ppca = ppca(X, dim)
print("Imputed Data (Dimensionality reduced):")
print(X_imputed_ppca)
```

```
warning: matrix is singular
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[68], line 37
     34     return Z  # Return the low-dimensional representation
     36 dim = 2
---> 37 X_imputed_ppca = ppca(X, dim)

     38 print("Imputed Data (Dimensionality reduced):")
     39 print(X_imputed_ppca)

Cell In[68], line 30, in ppca(X, dim, epsilon, max_iter)
     28     print("warning: matrix is singular")
     29     denominator += 1e-6 * np.eye(dim)
---> 30 W = np.linalg.inv(denominator) @ numerator  # Update W by getting the inverse of the denominator and so
on...
     32 if np.mean(np.linalg.norm(Z - Z_old, axis=1)) < epsilon:
     33     break

ValueError: matmul: Input operand 1 has a mismatch in its core dimension 0, with gufunc signature (n?,k),(k,m?)-
>(n?,m?) (size 20 is different from 2)
```

In [ ]:

c) MSE for PCA and PPCA

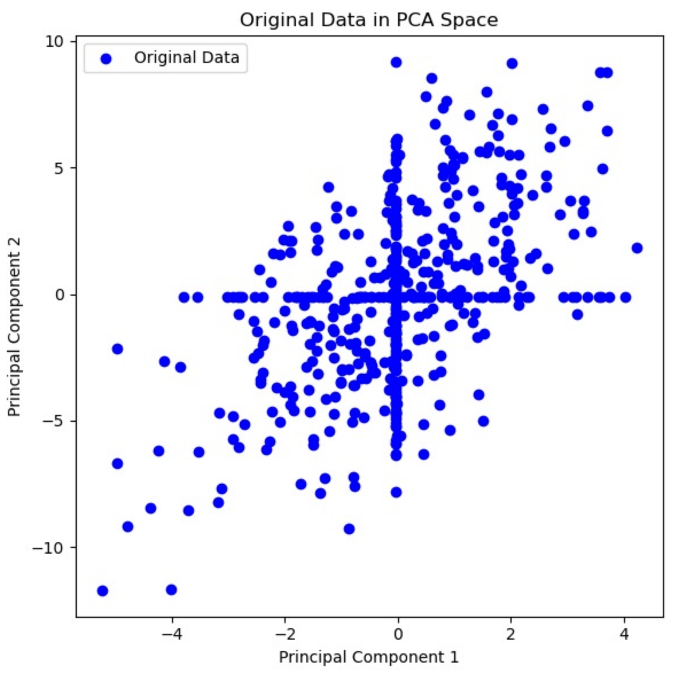
In [52]:
```python
mse = np.mean((X - X_imputed) ** 2)
print("Mean Squared Error (MSE) for PCA:", mse)
```

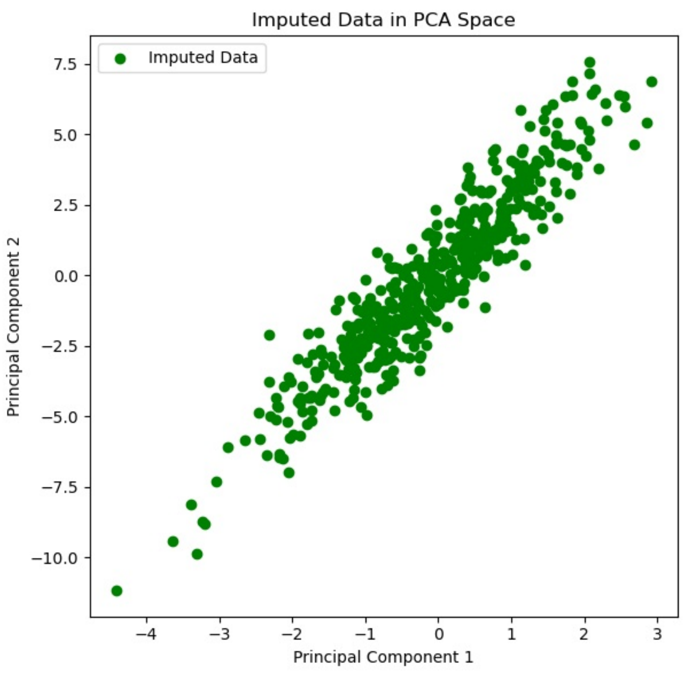
```
Mean Squared Error (MSE) for PCA: 2.1917232147460473
```

In [ ]:

d) Visualisation

In [67]:
```python
def pca1(X, rank):
    X = X.T  # Transpose for correct orientation
    U, S, Vt = np.linalg.svd(X, full_matrices=False)
    U_k = U[:, :rank]   # Truncate matrices for dimensionality reduction
    S_k = np.diag(S[:rank])  # Top K singular values
    Z = np.dot(U_k, S_k)
    return Z.T

def pca(X, rank):
    U, S, Vt = np.linalg.svd(X, full_matrices=False)

    U_d = U[:, :rank]  #truncate SVD
    S_d = np.diag(S[:rank])
    V_d = Vt[:rank, :]

    X_approx = np.dot(U_d, np.dot(S_d, V_d))

    return X_approx

def mean_impute(data):
    data_imputed = data.copy()
    col_mean = np.nanmean(data_imputed, axis=0)
    inds = np.where(np.isnan(data_imputed))
    data_imputed[inds] = np.take(col_mean, inds[1])
    return data_imputed

X_imputed = mean_impute(X)
# Perform PCA on both original and imputed data
K = 2
#D_dimred_original = pca(mean_impute(X), K)  # Original data (with imputation for PCA)
D_dimred_imputed = pca(X_imputed, K)  # Imputed data

fig, axes = plt.subplots(1, 2, figsize=(12, 6))

axes[0].scatter(X[:, 0], X[:, 1], color='blue', label='Original Data') # Original Data PCA scatter plot
axes[0].set_title("Original Data in PCA Space")
axes[0].set_xlabel("Principal Component 1")
axes[0].set_ylabel("Principal Component 2")
axes[0].legend()

axes[1].scatter(D_dimred_imputed[:, 0], D_dimred_imputed[:, 1], color='green', label='Imputed Data') # Imputed
axes[1].set_title("Imputed Data in PCA Space")
axes[1].set_xlabel("Principal Component 1")
axes[1].set_ylabel("Principal Component 2")
axes[1].legend()

plt.tight_layout()
plt.show()
```



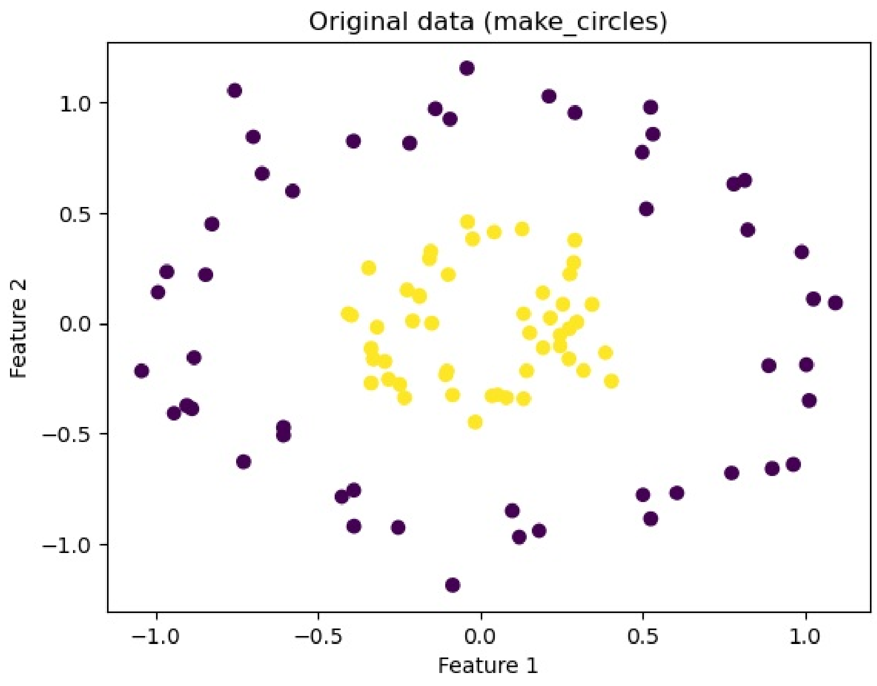


```
In [ ]:
```

2) Kernel PCA

a)

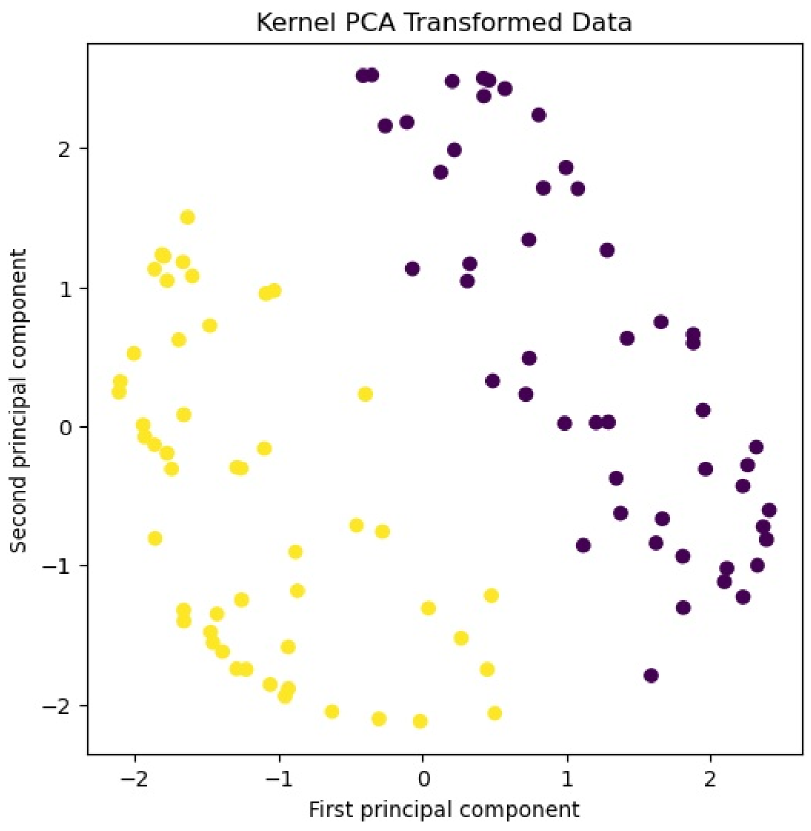
```
In [80]: from sklearn.datasets import make_circles
X, y = make_circles(n_samples=100, noise=0.1, factor=0.3)

plt.scatter(X[:, 0], X[:, 1], c=y, cmap='viridis') # Visualize the original dataset
plt.title('Original data (make_circles)')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.show()
```



b)

```
In [81]: def rbf_kernel(X, sigma=1.0): # computing RBF kernel
    pairwise_dists = np.sum(X**2, axis=1).reshape(-1, 1) + np.sum(X**2, axis=1) - 2 * np.dot(X, X.T)
    return np.exp(-pairwise_dists / (2 * sigma**2))

def kernel_pca(X, sigma=1.0, n_components=2):
    N = X.shape[0]
    K = rbf_kernel(X, sigma=sigma)  # computing the kernel matrix

    H = np.eye(N) - np.ones((N, N)) / N
    k_center = np.dot(np.dot(H, K), H) # centering kernel matrix
    eigenvalues, eigenvectors = np.linalg.eigh(k_center)  # eigenvalue decomposition

    sorted_idx = np.argsort(eigenvalues)[::-1]   # sorting eigenvalues, eigenvectors in desc order
    eigenvalues = eigenvalues[sorted_idx]
    eigenvectors = eigenvectors[:, sorted_idx]
    eigenvectors = eigenvectors[:, :n_components] # selecting the top n_components eigenvectors (principal compo
    projections = np.dot(k_center, eigenvectors) # Projecting data into the new space

    return projections, eigenvalues, eigenvectors
```

In [82]:
```
sigma = 0.5
n_components = 2  # aim is to project 2D
X_kpca, eigenvalues, eigenvectors = kernel_pca(X, sigma=sigma, n_components=n_components)

plt.figure(figsize=(6, 6))
plt.scatter(X_kpca[:, 0], X_kpca[:, 1], c=y, cmap='viridis')
plt.title('Kernel PCA Transformed Data')
plt.xlabel('First principal component')
plt.ylabel('Second principal component')
plt.show()
```



- Kernel PCA is useful for non-linear datasets like the make_circles dataset, traditional PCA might fail to capture the structure of the data.
- The scatter plot shows that Kernel PCA has successfully separated the two classes into regions

In [ ]:

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js