Principle Component Analysis

Dataset: Iris

- Load the dataset and perform necessary preprocessing steps, such as handling missing values, scaling, etc.
- Implement PCA from scratch using Python, NumPy, and Matplotlib, and apply it to the dataset.
- Use the scikit-learn library to apply PCA to the dataset and compare the results with the implementation from scratch.
- Visualize the results of PCA using Matplotlib or any other visualization library of your choice.
- Evaluate the performance of PCA by calculating the explained variance ratio for each principal component and selecting the appropriate number of principal components for the dataset.
- Use the selected number of principal components to reconstruct the original dataset and calculate the reconstruction error.
- Compare the results of PCA with and without dimensionality reduction using a classification algorithm of your choice, such as logistic regression, k-nearest neighbors, or support vector machines.

In [ ]:

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
from sklearn import datasets
```

In [ ]:

```
# Load the dataset
iris = datasets.load_iris()
df = pd.DataFrame(data=iris.data, columns=iris.feature_names)

# Add target and class to DataFrame
df['target'] = iris.target
df['class'] = df.target.apply(lambda x: iris.target_names[x])
df
```

Out[ ]:

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target	class
0	5.1	3.5	1.4	0.2	0	setosa
1	4.9	3.0	1.4	0.2	0	setosa
2	4.7	3.2	1.3	0.2	0	setosa
3	4.6	3.1	1.5	0.2	0	setosa
4	5.0	3.6	1.4	0.2	0	setosa
...	...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	2	virginica
146	6.3	2.5	5.0	1.9	2	virginica
147	6.5	3.0	5.2	2.0	2	virginica
148	6.2	3.4	5.4	2.3	2	virginica
149	5.9	3.0	5.1	1.8	2	virginica

150 rows × 6 columns

In [ ]:

```
# perform necessary preprocessing steps, such as handling missing values, scaling,
df.isnull().sum()
```

Out[ ]:

```
sepal length (cm)    0
sepal width (cm)     0
petal length (cm)    0
petal width (cm)     0
target               0
class                0
dtype: int64
```

In [ ]:

```
# Scaling
scaler = StandardScaler()
df_scaled = pd.DataFrame(scaler.fit_transform(df.drop(['target', 'class'], axis=1))
df_scaled['target'] = df['target']
df_scaled['class'] = df['class'] #target and class are not scaled, because they are
df_scaled
```

Out[ ]:

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target	class
0	-0.900681	1.019004	-1.340227	-1.315444	0	setosa
1	-1.143017	-0.131979	-1.340227	-1.315444	0	setosa
2	-1.385353	0.328414	-1.397064	-1.315444	0	setosa
3	-1.506521	0.098217	-1.283389	-1.315444	0	setosa
4	-1.021849	1.249201	-1.340227	-1.315444	0	setosa
...	...	...	...	...	...	...
145	1.038005	-0.131979	0.819596	1.448832	2	virginica
146	0.553333	-1.282963	0.705921	0.922303	2	virginica
147	0.795669	-0.131979	0.819596	1.053935	2	virginica
148	0.432165	0.788808	0.933271	1.448832	2	virginica
149	0.068662	-0.131979	0.762758	0.790671	2	virginica

150 rows × 6 columns

In [ ]:

```
# Implement PCA from scratch using Python, NumPy, and Matplotlib, and apply it to t
X = df_scaled.drop(['target', 'class'], axis=1).values
y = df_scaled['target'].values
```

In [ ]:

```
# Compute the covariance matrix
covariance_matrix = np.cov(X.T)
covariance_matrix
```

Out[ ]:

```
array([[ 1.00671141, -0.11835884,  0.87760447,  0.82343066],
       [-0.11835884,  1.00671141, -0.43131554, -0.36858315],
       [ 0.87760447, -0.43131554,  1.00671141,  0.96932762],
       [ 0.82343066, -0.36858315,  0.96932762,  1.00671141]])
```

In [ ]:

```
# Compute the eigenvalues and eigenvectors
eigenvalues, eigenvectors = np.linalg.eig(covariance_matrix)
eigenvalues
```

Out[ ]:

```
array([2.93808505, 0.9201649 , 0.14774182, 0.02085386])
```

In [ ]:

```
eigenvectors
```

Out[ ]:

```
array([[ 0.52106591, -0.37741762, -0.71956635,  0.26128628],
       [-0.26934744, -0.92329566,  0.24438178, -0.12350962],
       [ 0.5804131 , -0.02449161,  0.14212637, -0.80144925],
       [ 0.56485654, -0.06694199,  0.63427274,  0.52359713]])
```

In [ ]:

```
# Sort the eigenvalues and corresponding eigenvectors
eigenvalue_indices = np.argsort(eigenvalues)[::-1]
sorted_eigenvalues = eigenvalues[eigenvalue_indices]
sorted_eigenvectors = eigenvectors[:, eigenvalue_indices]
```

In [ ]:

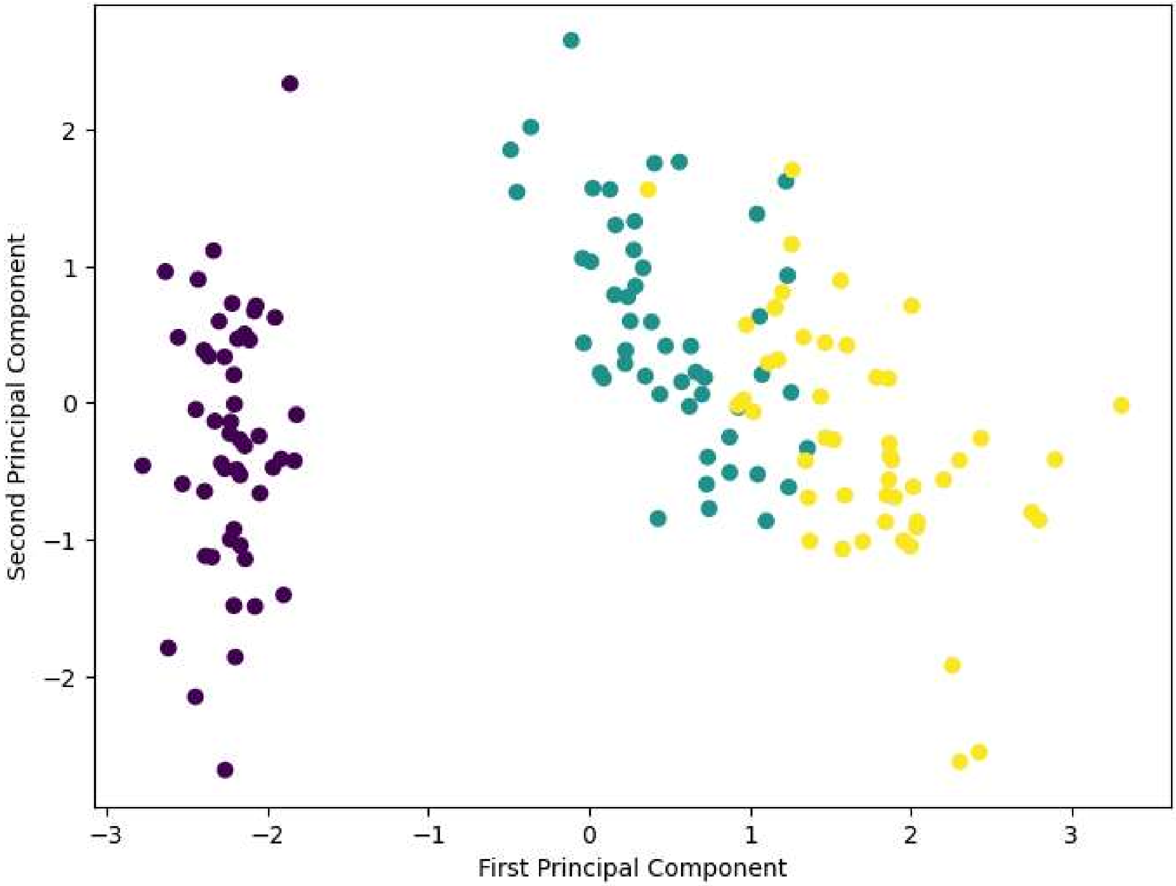
```
# Select the top 2 eigenvectors
eigenvectors_subset = sorted_eigenvectors[:, :2]
eigenvectors_subset
```

Out[ ]:

```
array([[ 0.52106591, -0.37741762],
       [-0.26934744, -0.92329566],
       [ 0.5804131 , -0.02449161],
       [ 0.56485654, -0.06694199]])
```

In [ ]:
```
# Transform the data
X_pca = X.dot(eigenvectors_subset)
```

In [ ]:
```
# Visualize the results of PCA using Matplotlib
plt.figure(figsize=(8,6))
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=df_scaled['target'])
plt.xlabel('First Principal Component')
plt.ylabel('Second Principal Component')
plt.show()
```



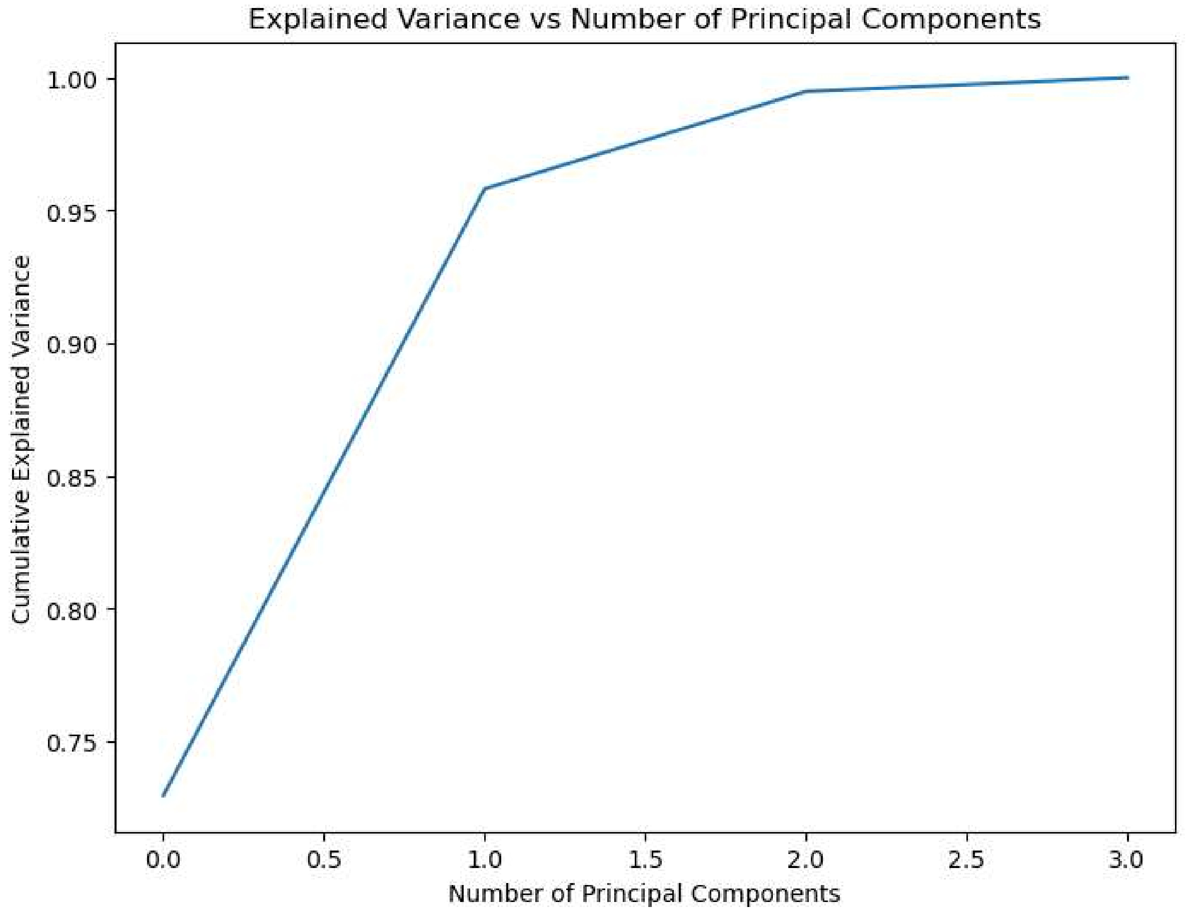
In [ ]:
```
# Evaluate the performance of PCA by calculating the explained variance ratio for e
# Calculate the explained variance
explained_variance = sorted_eigenvalues / np.sum(sorted_eigenvalues)
explained_variance
```

Out[ ]:
```
array([0.72962445, 0.22850762, 0.03668922, 0.00517871])
```

In [ ]:
```
# Calculate the cumulative explained variance
cumulative_explained_variance = np.cumsum(explained_variance)
cumulative_explained_variance
```

Out[ ]:
```
array([0.72962445, 0.95813207, 0.99482129, 1.        ])
```

In [ ]:
```
# Plot the cumulative explained variance
plt.figure(figsize=(8,6))
plt.plot(range(len(cumulative_explained_variance)), cumulative_explained_variance)
plt.xlabel('Number of Principal Components')
plt.ylabel('Cumulative Explained Variance')
plt.title('Explained Variance vs Number of Principal Components')
plt.show()
```



In [ ]:
```
# Select the appropriate number of principal components
n_components = np.argmax(cumulative_explained_variance > 0.95) + 1
print(f"The appropriate number of principal components is {n_components}")
```

```
The appropriate number of principal components is 2
```

In [ ]:
```
# Use the selected number of principal components to reconstruct the original datas
pca = PCA(n_components=n_components)
X_pca = pca.fit_transform(X)
X_reconstructed = pca.inverse_transform(X_pca)
```

In [ ]:
```
# Calculate the reconstruction error
reconstruction_error = np.mean((X - X_reconstructed) ** 2)
print(f"The reconstruction error is {reconstruction_error}")
```

```
The reconstruction error is 0.041867927999983595
```

In [ ]:
```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
```

In [ ]:
```
# Compare the results of PCA with dimensionality reduction and without dimensionali
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train)
y_pred = knn.predict(X_test)
print("Accuracy without dimensionality reduction: ", accuracy_score(y_test, y_pred)
```

```
Accuracy without dimensionality reduction:  0.9333333333333333
```

In [ ]:
```
# Split the PCA transformed data into training and testing sets
X_train_pca, X_test_pca, y_train_pca, y_test_pca = train_test_split(X_pca, y, test_

# Train the KNN model with the PCA transformed training set
```

```
knn.fit(X_train_pca, y_train_pca)

# Make predictions with the PCA transformed testing set
y_pred_pca = knn.predict(X_test_pca)

print("Accuracy with dimensionality reduction: ", accuracy_score(y_test_pca, y_pred
```

```
Accuracy with dimensionality reduction:  0.9666666666666667
```