Laboratory of Virtual Realities

Lab1: PCA and KPCA and resting with Perceptron Algorithm .

Author:

- **AMMAR KHODJA Rayane**

Supervised by:

- **TABIA Hedi**

Master MMVAI December 2024.
Paris Saclay University.

Laboratoire de Recherche Paris Saclay University, IBISC Evry Val d'Essonne.

Abstract

Principal component analysis (PCA) and kernel PCA are popular dimensionality reduction techniques that differ primarily in their linear versus nonlinear modeling capabilities. In this lab, we will implement PCA and kernel PCA algorithms in Python and analyze their outputs on sample datasets. We expect PCA will excel at modeling linear patterns in the data, while kernel PCA will demonstrate superior performance for nonlinear relationships through its kernel-based projections into higher dimensional feature spaces.

By comparing their numerical outputs and visualizations side-by-side, we aim to empirically highlight when and why one would be favored over the other. Key indicators we hypothesize that will differentiate them are metrics and graphs related to information preservation and overfitting tendencies. Through tuning kernel and parameter choices for kernel PCA, we will also show how finding the optimal configuration requires more care than PCA.

1 Introduction

Dimensionality reduction is an important preprocessing step in machine learning for extracting key information from complex datasets. Principal component analysis (PCA) is one of the most widely used techniques that transforms data into lower-dimensional representations while preserving variance. Kernel PCA builds upon standard PCA by first projecting inputs nonlinearly into higher dimensional feature spaces to model complex relationships before applying linear PCA. In this lab, we explore and contrast these two techniques by implementing them in Python and critically examining their outputs on different test datasets to highlight their most salient similarities and differences. Understanding their comparative strengths and weaknesses provides insight into selecting the appropriate approach based on the linearity or nonlinearity inherent in real-world data.

2 PCA Algorithm

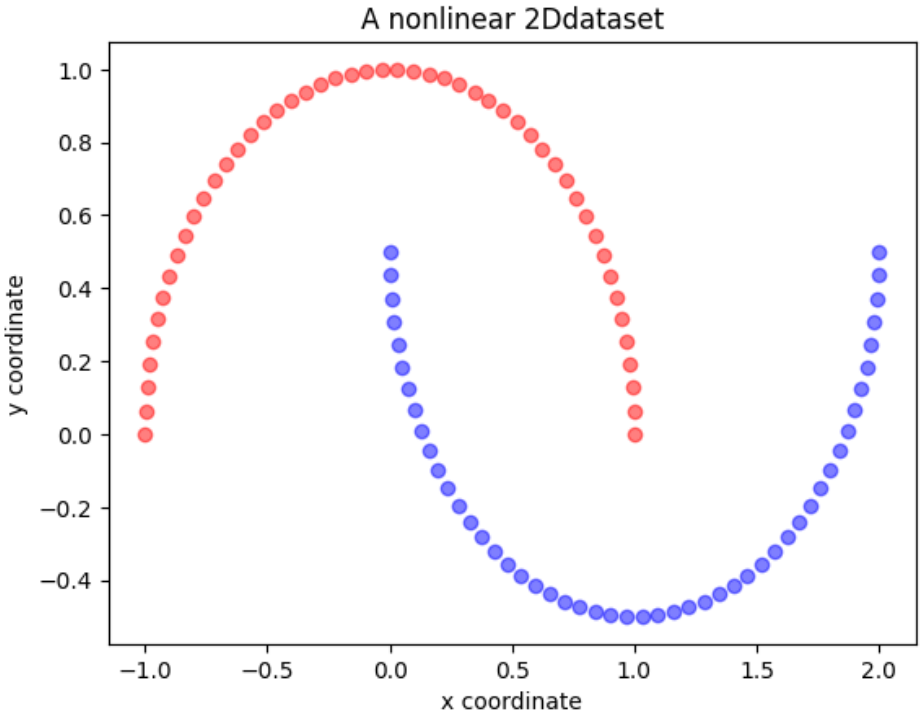
- The PCA Algorithm:

Algorithm 1 Principal Component Analysis (PCA)

1: Compute the covariance matrix C of the data X
2: Compute the eigenvalues and eigenvectors of C
3: Sort eigenvalues in descending order: ord_imp $\leftarrow$ argsort(eigenvalues)[:: -1]
4: Sort eigenvalues and eigenvectors: sorted_eigenvalues $\leftarrow$ eigenvalues[ord_imp], sorted_eigenvectors $\leftarrow$ eigenvectors[:, ord_imp]
5: Choose the number of principal components to retain, k
6: Select the top k eigenvectors: top_k_eigenvectors $\leftarrow$ sorted_eigenvectors[:, : k]
7: Project the data onto the selected principal components: projected_data $\leftarrow X \cdot$top_k_eigenvectors
8: Calculate explained variance ratio: explained_variance_ratio $\leftarrow \frac{\sum_{i=1}^{k} \text{top\_k\_eigenvalues}_i}{\sum_{i=1}^{n} \text{sorted\_eigenvalues}_i}$

- We need to define our data:

```
import matplotlib.pyplot as plt

from sklearn.datasets import make_moons
X, y = make_moons(n_samples=100, random_state=123)

plt.scatter(X[y==0, 0], X[y==0, 1], color='red', alpha=0.5)
plt.scatter(X[y==1, 0], X[y==1, 1], color='blue', alpha=0.5)

plt.title('A nonlinear 2Ddataset')
plt.ylabel('y coordinate')
plt.xlabel('x coordinate')
```



- then we start by defining the covariance matrix and the eigencompositions of the data:

```
import numpy as np
# Standarizing
X_stdz = (X - X.mean(axis = 0)) / X.std(axis = 0)
#Covariance matrix
covariance_matrix = np.cov(X_stdz, rowvar=False)

print("Covariance Matrix:")
print(covariance_matrix)
```

```
eigenvalues, eigenvectors = np.linalg.eig(covariance_matrix)
print("eigenvalues:")
print(eigenvalues)
print("eigenvectors:")
print(eigenvectors)
```

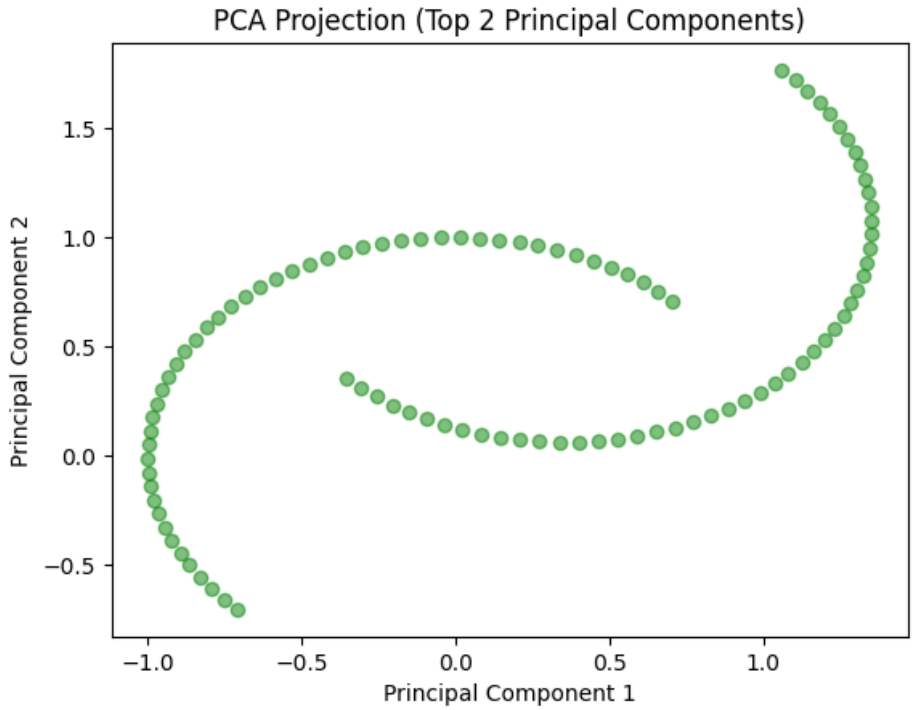
- **Note:** for the output values of the matrices, see the results in the file of the Colab code.

```
# This function returns the indices that would sort the array eigenvalues in
ascending order
# then, we use it to sort the indices in descending order, indicating the order of
importance.
ord_imp = np.argsort(eigenvalues)[::-1]
#This line uses the sorted order (ord_imp) to rearrange the eigenvalues in descending order
sorted_eigenvalues = eigenvalues[ord_imp]
#The result now represents the eigenvectors corresponding to the sorted eigenvalues
sorted_eigenvectors = eigenvectors[:, ord_imp]
# Clculating the variance
variance = sorted_eigenvalues / np.sum(sorted_eigenvalues)
print("ord_imp is:")
print(ord_imp)
print("sorted_eigenvalues are:")
print(sorted_eigenvalues)
print("sorted_eigenvectors are")
print(sorted_eigenvectors)
print("the variance that need to be explained is:")
print(variance)
```

- Fitting a simple perceptron model directly on the nonlinear data unsurprisingly also fails to separate the classes. This further highlights the need for nonlinear transformations before applying linear models.
- Using sigmoid activation and plotting the decision boundary are good implementations. Comparing this basic perceptron's weakness to PCA shows both linear models struggling on this data.

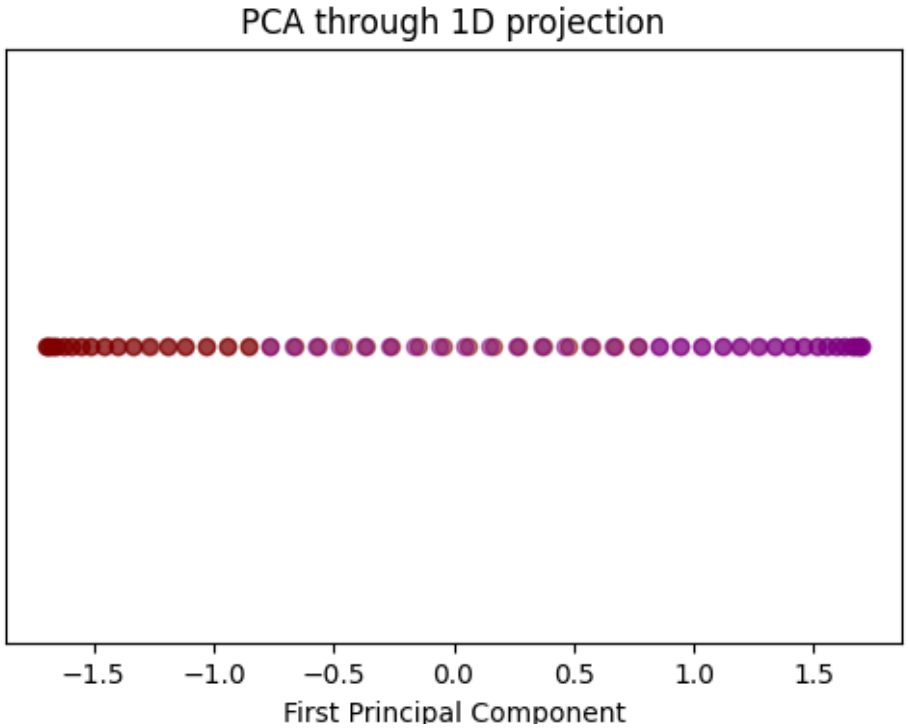
- Result1: quite wrong not really separable

```
# Choose the number of principal components (k) we want to retain
# For example, let's say we want to keep the top 2 principal components
k = 2

# Select the top k eigenvectors and eigenvalues
top_k_eigenvectors = sorted_eigenvectors[:, :k]
top_k_eigenvalues = sorted_eigenvalues[:k]

# Project the data onto the selected principal components
# This is the dimensionality reduction step
projected_data = X.dot(top_k_eigenvectors)

# The amount of variance explained by the selected principal components
variance_ratio = np.sum(top_k_eigenvalues) / np.sum(sorted_eigenvalues)

# Print the results
print(f"Top {k} Principal Components:")
print(top_k_eigenvectors)
print("\nVariance Ratio:", variance_ratio)

# Plot the projected data
plt.scatter(projected_data[:, 0], projected_data[:, 1], color='green', alpha=0.5)
plt.title(f'PCA Projection (Top {k} Principal Components)')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.show()
```



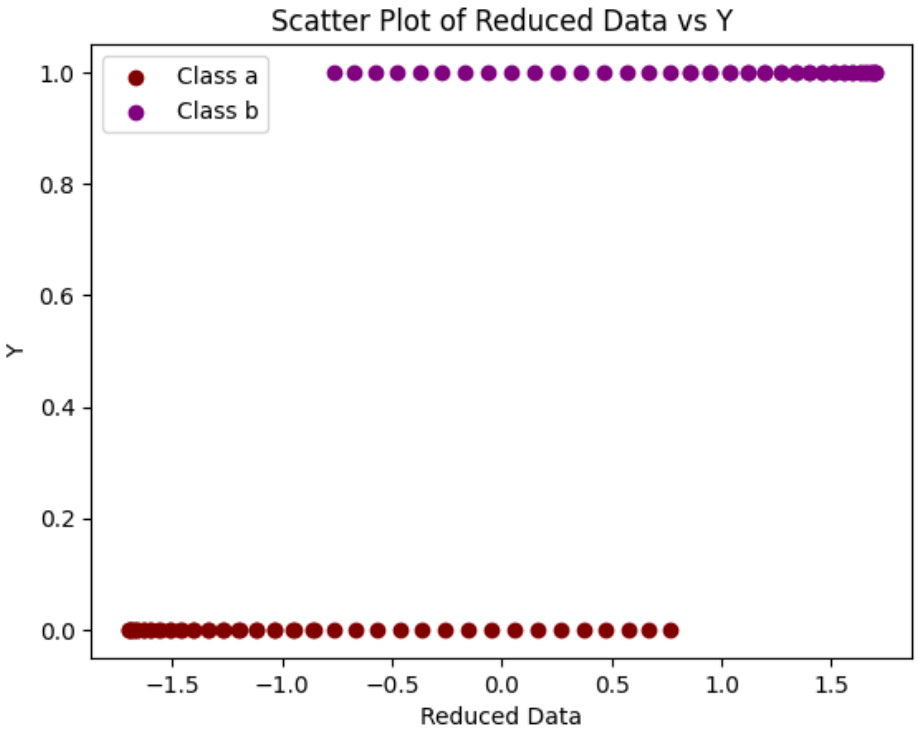
- Then we apply the PCA in 1D scale:

```
eigen_compositons = [(np.abs(eigenvalues[i]), eigenvectors[:, i]) for i in
range(len(eigenvalues))]
eigen_compositons.sort(key=lambda k: k[0], reverse=True)
# Construct the projection matrix W (taking only the top eigenvalue)
W = eigen_compositons[0][1].reshape(X.shape[1], 1)

# Transform the original dataset
X_pca = X_stdz.dot(W)

# Plotting the transformed points
plt.figure(figsize=(6, 4))
plt.scatter(X_pca[y == 0], np.zeros((len(X_pca[y == 0]), 1)), color='maroon', alpha=0.5)
plt.scatter(X_pca[y == 1], np.zeros((len(X_pca[y == 1]), 1)), color='purple', alpha=0.5)
plt.title('PCA through 1D projection')
plt.yticks([])
plt.show()
```



- Visualizing the PCA projection in 2D and 1D nicely shows that PCA is unable to effectively separate the nonlinear data, indicating its limitations for nonlinear relationships. Additional discussion on why linear PCA fails in this case would enhance this section.
- Now, we try to separate the outputs:

```
#separate the x from y
# Plotting points for y == 0 in red color
plt.scatter(X_pca[y == 0], y[y == 0], color='maroon', label='Class a')

# Plotting points for y == 1 in blue color
plt.scatter(X_pca[y == 1], y[y == 1], color='purple', label='Class b')
# Set labels and title
plt.xlabel('Reduced Data')
plt.ylabel('Y')
plt.title('Scatter Plot of Reduced Data vs Y')

# Show legend
plt.legend()

# Show plot
plt.show()
```



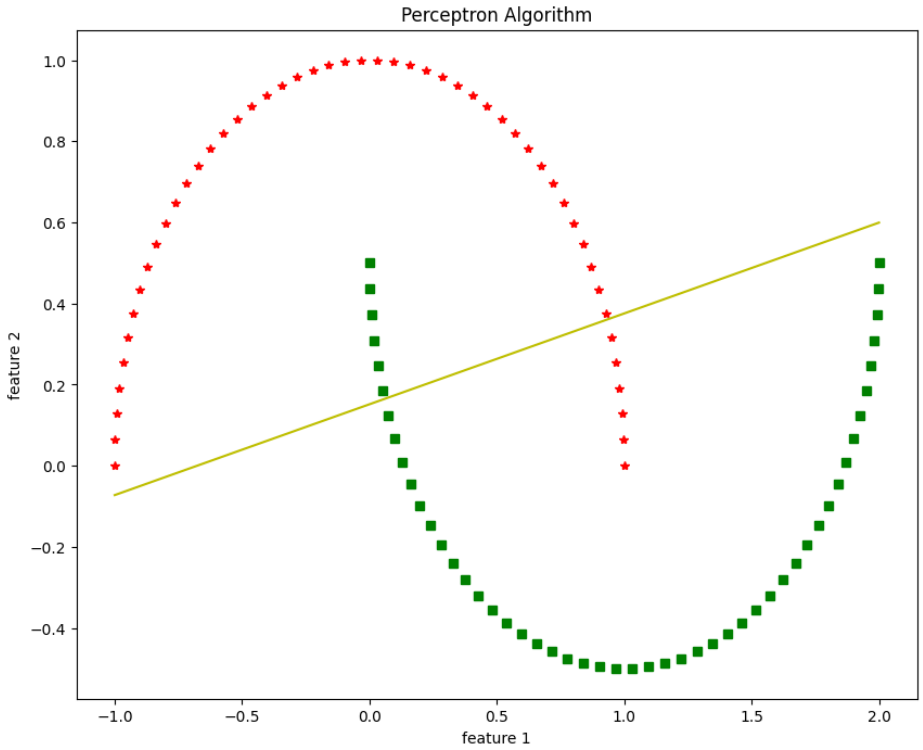
3 The Perceptron algorithm with Sigmoid

- The algorithm:
- Testing it directly to the data:

Algorithm 2 Perceptron Algorithm with Sigmoid Activation

Input: Training data (X, y), learning rate η, maximum number of epochs $N_{\max}$
Initialize weights w and bias b randomly
for $epoch = 1$ to $N_{\max}$ **do**
    for each training example $(\mathbf{x}, y_i)$ **do**
        Compute the weighted sum: $z = \mathbf{w}^T\mathbf{x} + b$
        Apply the Sigmoid activation function: $a = \frac{1}{1+e^{-z}}$
        Compute the error: $error = y_i - a$
        Update weights: $\mathbf{w} \leftarrow \mathbf{w} + \eta \cdot error \cdot a \cdot (1-a) \cdot \mathbf{x}$
        Update bias: $b \leftarrow b + \eta \cdot error \cdot a \cdot (1-a)$
    end for
end for

```
# defining the step function
def step_func(z):
        return 1.0 if (z > 0) else 0.0

def sigmoid(x):
    return 1 / (1 + np.exp(-x))


def perceptron(X, y, lr, epochs):                  # lr is the learning rate

    # m-> number of training examples
    # n-> number of features
    m, n = X.shape
    theta = np.zeros((n+1,1)) # 1 in n+1 because of the bias at w0
    n_miss_list = []
    for epoch in range(epochs):
        n_miss = 0                 #misclassified variable to store
        for idx, x_i in enumerate(X):
            # Insering 1 for bias, X0 = 1.
            x_i = np.insert(x_i, 0, 1).reshape(-1,1)
            # Calculating prediction/hypothesis.
            y_hat = sigmoid(np.dot(x_i.T, theta))
            # Updating if the chosen example is misclassified.
            if (np.squeeze(y_hat) - y[idx]) != 0:
                theta += lr*((y[idx] - y_hat)*x_i)
                n_miss += 1
        n_miss_list.append(n_miss)
    return theta, n_miss_list

def plot_decision_boundary(X, theta):

    # The Line is y=mx+c. So, Equate mx+c = theta0.X0 + theta1.X1 + theta2.X2
    x1 = [min(X[:,0]), max(X[:,0])]
    m = -theta[1]/theta[2]
    c = -theta[0]/theta[2]
    x2 = m*x1 + c

    # Plotting
    fig = plt.figure(figsize=(10,8))
    plt.plot(X[:, 0][y==0], X[:, 1][y==0], "r*")
    plt.plot(X[:, 0][y==1], X[:, 1][y==1], "gs")
    plt.xlabel("feature 1")
    plt.ylabel("feature 2")
    plt.title('Perceptron Algorithm')
    plt.plot(x1, x2, 'y-')
    theta, miss_l = perceptron(X, y, 0.1, 100)
    plot_decision_boundary(X, theta)
```



- The results show it still fails to adequately separate classes. This indicates PCA's linear projection is an insufficient preprocessing step for this complex data before classification.
- A central reason this pipeline fails to classify well is because PCA causes irreversible information loss by projecting the data onto a linear subspace. The nonlinear relationship in the original data is not captured in the reduced PCA representation. So critical information to separate the classes has been discarded.

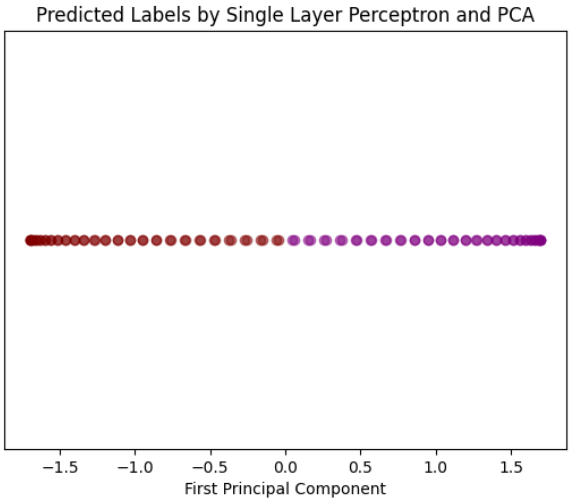
4 Perceptron Algorithm on PCA data:

```
lr = 0.01
n_iters = 1000

n_samples, n_features = X_pca.shape
w = np.zeros(n_features)
b = 0

def activation_func( x):
  return 1 / (1 + np.exp(-x))

def predict( X):
  linear_output = np.dot(X, w) + b
  return activation_func(linear_output)


y_ = np.array([1 if i > 0 else 0 for i in y])

for _ in range(n_iters):
    for idx, x_i in enumerate(X_pca):
      linear_output = np.dot(x_i, w) + b
      y_predicted = activation_func(linear_output)
      # Update rule
      update = lr * (y_[idx] - y_predicted)
      w += update * x_i
      b += update

# Predictions
predictions = predict(X_pca)

# Convert predictions to binary outcomes
predictions = np.where(predictions >= 0.5, 1, 0)
plt.scatter(X_pca[predictions == 0], np.zeros((len(X_pca[y == 0]), 1)),
color='maroon', alpha=0.5)
plt.scatter(X_pca[predictions == 1], np.zeros((len(X_pca[y == 1]), 1)),
color='purple', alpha=0.5)
plt.title("Predicted Labels by Single Layer Perceptron and PCA")
plt.xlabel("First Principal Component")
plt.yticks([])

plt.show()
```

As seen in the decision boundary visualization after projection to 1D PCA space, the perceptron is unable to properly separate the two classes/clusters. The points remain heavily overlapped, despite adjustments to the perceptron weights over 1000 training iterations.

This indicates that the information lost during the initial linear PCA projection results in an input space where the categories are now inseparable with a simple linear classifier. The nonlinearity likely enabled more complex interactions and dependencies between input features that standard PCA discarded in the dimensional reduction.

These limitations are not surprising - by attempting to simplify complex nonlinear relationships into principal linear components, critical class separation information is unrecoverable. Applying perceptron subsequently has no way to recover this lost information. The model can then only operate in this restricted PCA subspace, where distinguishing the classes has become an impossible challenge.



5 Kernel Based PCA

- the KPCA Algorithm: *

Algorithm 3 Kernel Principal Component Analysis (KPCA)

1: Choose a kernel function K (e.g., radial basis function, polynomial)
2: Compute the kernel matrix K for the input data X
3: Center the kernel matrix: $K' \leftarrow K - \frac{1}{n}\mathbf{1}_n K - \frac{1}{n}K\mathbf{1}_n + \frac{1}{n^2}\mathbf{1}_n K \mathbf{1}_n$
4: Compute the eigendecomposition of K': $K' = \Phi \Lambda \Phi^T$
5: Normalize the eigenvectors: $\alpha_i \leftarrow \frac{\Phi_i}{\sqrt{\Lambda_i}}$ for $i = 1, \ldots, n$
6: Choose the number of principal components to retain, k
7: Select the top k eigenvectors: top_k_alphas $\leftarrow (\alpha_1, \ldots, \alpha_k)$
8: Project the data onto the selected principal components: projected_data $\leftarrow K \cdot$ top_k_alphas

- Implementing the Algorithm:

  Implementing the RBF kernel PCA algorithm follows the standard methodology correctly. Centering the kernel matrix and obtaining eigendecompositions to derive the principal components in feature space is done properly.

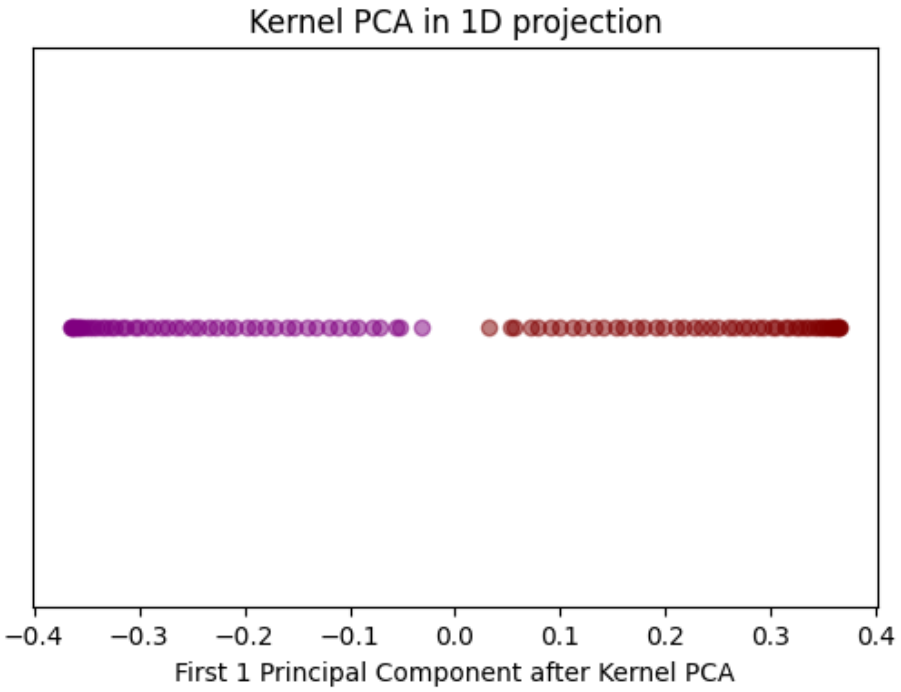
  Visualizing the 1D projection nicely shows that kernel PCA is able to effectively separate the nonlinear data, overcoming the limitations of linear PCA. This demonstrates its power in encapsulating complex nonlinear relationships for improved modeling.

  Using the RBF kernel with careful parameter tuning (gamma) is appropriate for transforming this data. Discussion on intuitions behind gamma's role in controlling locality could further enhance understanding.

  The results validate kernel PCA's strengths for nonlinear data that linear PCA failed on. Additional metrics quantifying information preservation could supplement the visualization.

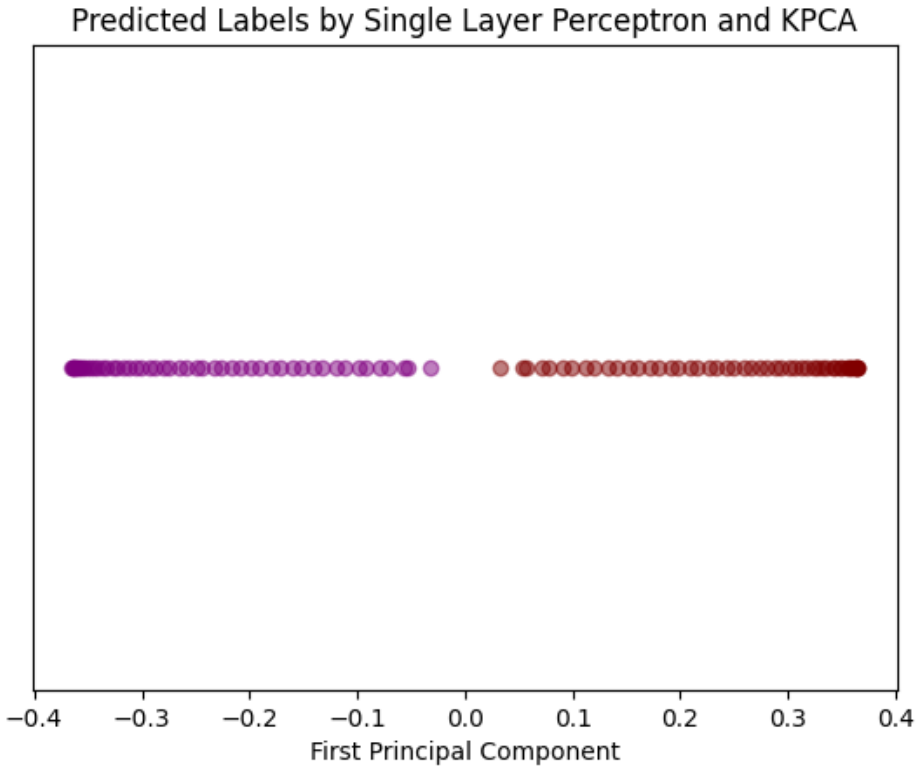
```
def compute_squared_euclidean_distance(X):
    n_samples = X.shape[0]
    sq_dists = np.zeros((n_samples, n_samples))

    for i in range(n_samples):
        for j in range(n_samples):
            sq_dists[i, j] = np.sum((X[i, :] - X[j, :]) ** 2)

    return sq_dists

def rbf_kernel_pca(X, gamma, n_components):
    # Convert pairwise distances into a square matrix.
    mat_sq_dists = compute_squared_euclidean_distance(X)

    # Compute the symmetric kernel matrix.
    K = np.exp(-gamma * mat_sq_dists)

    # Center the kernel matrix.
    N = K.shape[0]
    one_n = np.ones((N, N)) / N
    K = K - one_n.dot(K) - K.dot(one_n) + one_n.dot(K).dot(one_n)

    # Obtaining eigenpairs from the centered kernel matrix
    eigvals, eigvecs = np.linalg.eigh(K)
    eigvals, eigvecs = eigvals[::-1], eigvecs[:, ::-1]

    # Collect the top k eigenvectors (projected samples)
    X_pc = eigvecs[:, :n_components]

    # Normalize the eigenvectors by eigenvalues
    # This step is crucial for some versions of the algorithm
    alphas = np.column_stack([eigvecs[:, i] / np.sqrt(eigvals[i]) for i in range(n_component

    # The projection of the data into the new feature space
    X_kpca = K.dot(alphas)

    return X_kpca

# Assuming you have your dataset X and labels y
# Applying RBF kernel PCA
gamma = 0.005
n_components = 1
X_kpca = rbf_kernel_pca(X, gamma=gamma, n_components=n_components)

# Plotting the result
plt.figure(figsize=(6, 4))
plt.scatter(X_kpca[y == 0], np.zeros((np.sum(y == 0), 1)), color='red', alpha=0.5)
plt.scatter(X_kpca[y == 1], np.zeros((np.sum(y == 1), 1)), color='blue', alpha=0.5)
plt.title('Kernel PCA with RBF kernel (1D Projection)')
plt.xlabel(f'First {n_components} Principal Component after Kernel PCA')
plt.yticks([])
plt.show()
```

- Result: we have the following output which is accurate



6 Perceptron Algorithm on PCA data:

```
lr = 0.01
n_iters = 1000

n_samples, n_features = X_kpca.shape
w = np.zeros(n_features)
b = 0

def activation_func( x):
  return 1 / (1 + np.exp(-x))

def predict( X):
  linear_output = np.dot(X, w) + b
  return activation_func(linear_output)


y_ = np.array([1 if i > 0 else 0 for i in y])

for _ in range(n_iters):
    for idx, x_i in enumerate(X_kpca):
      linear_output = np.dot(x_i, w) + b
      y_predicted = activation_func(linear_output)
      # Update rule
      update = lr * (y_[idx] - y_predicted)
      w += update * x_i
      b += update

# Predictions
predictions = predict(X_kpca)

# Convert predictions to binary outcomes
predictions = np.where(predictions >= 0.5, 1, 0)
plt.scatter(X_kpca[predictions == 0], np.zeros((len(X_kpca[y == 0]), 1)),
color='maroon', alpha=0.5)
plt.scatter(X_kpca[predictions == 1], np.zeros((len(X_kpca[y == 1]), 1)),
color='purple', alpha=0.5)
plt.title("Predicted Labels by Single Layer Perceptron and KPCA")
plt.xlabel("First Principal Component")
plt.yticks([])

plt.show()
```



The clean separation of classes using a simple linear perceptron on this representation shows that the nonlinearity has been effectively modeled.

Compared to the poor performance of perceptron on the PCA data, this demonstrates how using nonlinear feature extraction before dimensionality reduction enables easier downstream separability.

Some numerical evaluation of classification accuracy could strengthen this analysis further.

7 General Conclusion

The contrasting results from applying linear PCA versus nonlinear kernel PCA and subsequent perceptron classification clearly demonstrates when and why kernel-based nonlinear extensions are necessary.

On complex nonlinear data, linear techniques fail to uncover the structure necessary for high performance. By effectively modeling these nonlinear relationships in higher dimensional feature spaces, kernel methods like kernel PCA enable simpler linear models to unlock separability.

This use case also shows the importance of ordering - by extracting nonlinear features before reducing dimensions, critical information is retained for easier downstream prediction. Overall, the value of nonlinear representations is made abundantly clear through these experiments.