PCA, LDA, Kernel PCA, and t-SNE/UMAPs

Machine Learning for Engineering Applications

Fall 2023

Data Compression

- **Principal Component Analysis (PCA)**
  - Unsupervised data compression
- **Linear Discriminant Analysis (LDA)**
  - Supervised dimensionality reduction
- **Kernel Principal Component Analysis (KPCA)**
  - Non-linear data compression approach

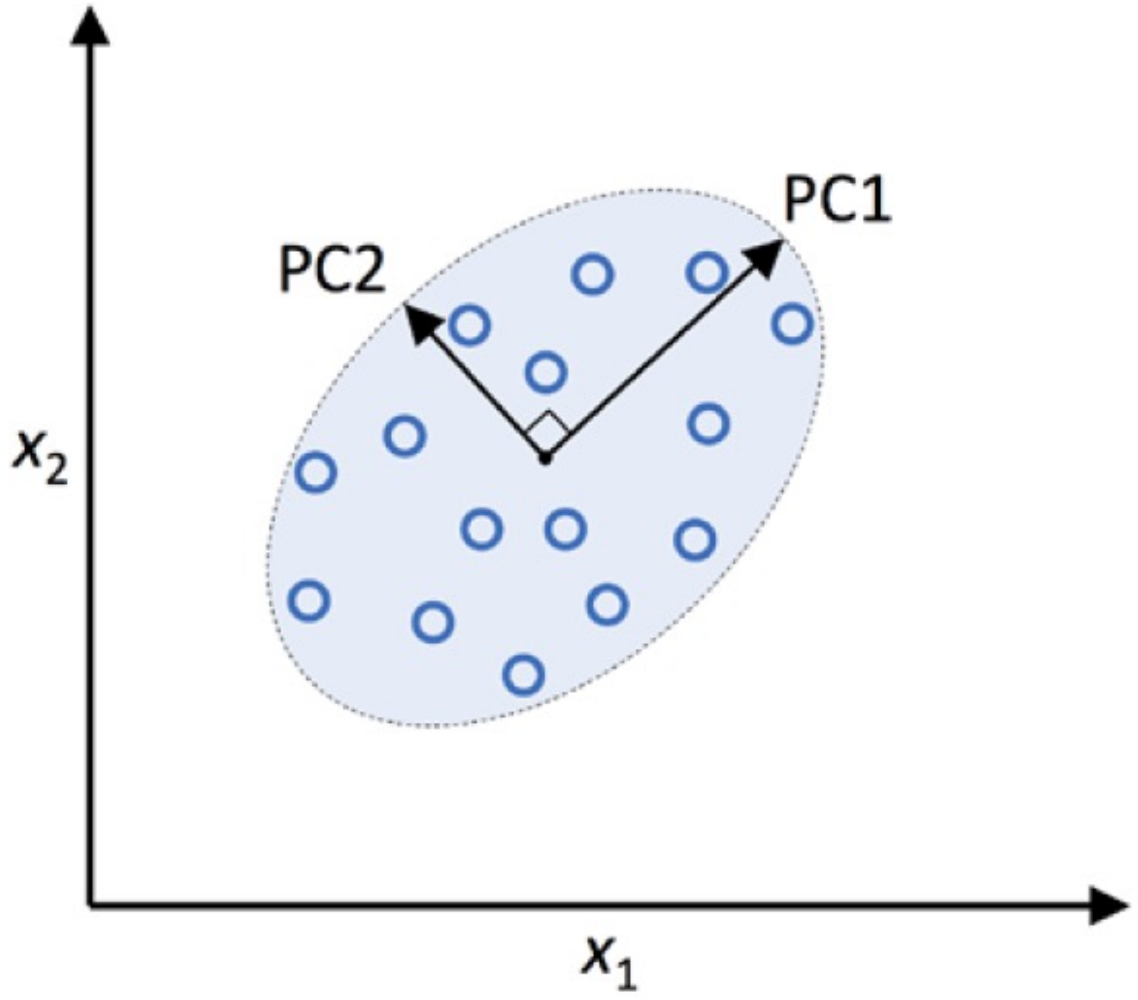
PCA

- In the context of dimensionality reduction:
  - feature extraction is the approach to data compression
  - **Goal**: maintaining most of the relevant information

- **PCAs** are widely used when you have a common problem: **too many features**!!!

- This will force you to perform a dimensional reduction without losing the information of all of the dataset

- A transformation helps to "*encapsulate*" the meaning of the data
- The PC components help to keep the integrity of the reduction of features
- The PCs need to be orthogonal to obtain a proper transformation of the dataset as its being reduced



- Original feature space
- Transformation operation that consider the reduction of a *d-dimensional* space to a *k-dimensional* space
- The compressed feature space

$$\boldsymbol{x} = [x_1, x_2, \ldots, x_d], \quad \boldsymbol{x} \in \mathbb{R}^{d}$$

$$\downarrow \boldsymbol{xW}, \quad \boldsymbol{W} \in \mathbb{R}^{d \times k}$$

$$\mathbf{z} = [z_1, z_2, \ldots, z_k], \quad \mathbf{z} \in \mathbb{R}^{k}$$

7-STEPS to PCA

1. Standardize the *d*-dimensional dataset.

2. Construct the covariance matrix.

3. Decompose the covariance matrix into its eigenvectors and eigenvalues.

4. Sort the eigenvalues by decreasing order to rank the corresponding eigenvectors.

5. Select *k* eigenvectors which correspond to the *k* largest eigenvalues, where *k* is the dimensionality of the new feature subspace ($k \leq d$).

6. Construct a projection matrix **W** from the "top" *k*-eigenvectors.

7. Transform the d-dimensional input dataset **X** using the projection matrix W to obtain the new *k*-dimensional feature subspace.

- Wines!!!
- The dataset has 13 features….who knew…
- Do you need all the features?
- Which features are more important?
- What if you do not know anything about wines?

1. Standardize the *d*-dimensional dataset.

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

```
df_wine =
pd.read_csv('https://archive.ics.uci.edu/ml/'machine-
learning-databases/wine/wine.data’, header=None)
X, y = df_wine.iloc[:, 1:].values, df_wine.iloc[:, 0].values
X_train, X_test, y_train, y_test =
     ... train_test_split(X, y, test_size=0.3, stratify=y,
     ... random_state=0)
```

```
sc = StandardScaler()
X_train_std = sc.fit_transform(X_train)
X_test_std = sc.transform(X_test)
```

2. Construct the covariance matrix.

$$\sigma_{jk} = \frac{1}{n}\sum_{i=1}^{n}\left(x_{\underline{j}}^{(i)} - \mu_{\underline{j}}\right)\left(x_{\underline{k}}^{(i)} - \mu_{\underline{k}}\right)$$

What's *n*?

What's *j* & *k*?

What's *i*?

$$\Sigma = \begin{bmatrix} \sigma_1^2 & \sigma_{12} & \sigma_{13} \\ \sigma_{21} & \sigma_2^2 & \sigma_{23} \\ \sigma_{31} & \sigma_{32} & \sigma_3^2 \end{bmatrix}$$

Covariance matrix for a 3-feature dataset

3. Decompose the covariance matrix into its eigenvectors and eigenvalues.

- The eigenvectors become the Principal Components of your dataset
- The eigenvalues are the level of importance to their corresponding eigenvectors
- Review: Eigenvectors must satisfy, $\Sigma \boldsymbol{v} = \lambda \boldsymbol{v}$

3. Decompose the covariance matrix into its eigenvectors and eigenvalues

```
sc = StandardScaler()
X_train_std = sc.fit_transform(X_train)
X_test_std = sc.transform(X_test)

import numpy as np
cov_mat = np.cov(X_train_std.T)
eigen_vals, eigen_vecs = np.linalg.eig(cov_mat)
print('\nEigenvalues \n%s' % eigen_vals)
```

```
Eigenvalues
[ 4.84274532 2.41602459 1.54845825 0.96120438 0.84166161
0.6620634    0.51828472 0.34650377 0.3131368  0.10754642
0.21357215   0.15362835 0.1808613 ]
```

4. Sort the eigenvalues by decreasing order to rank the corresponding eigenvectors.

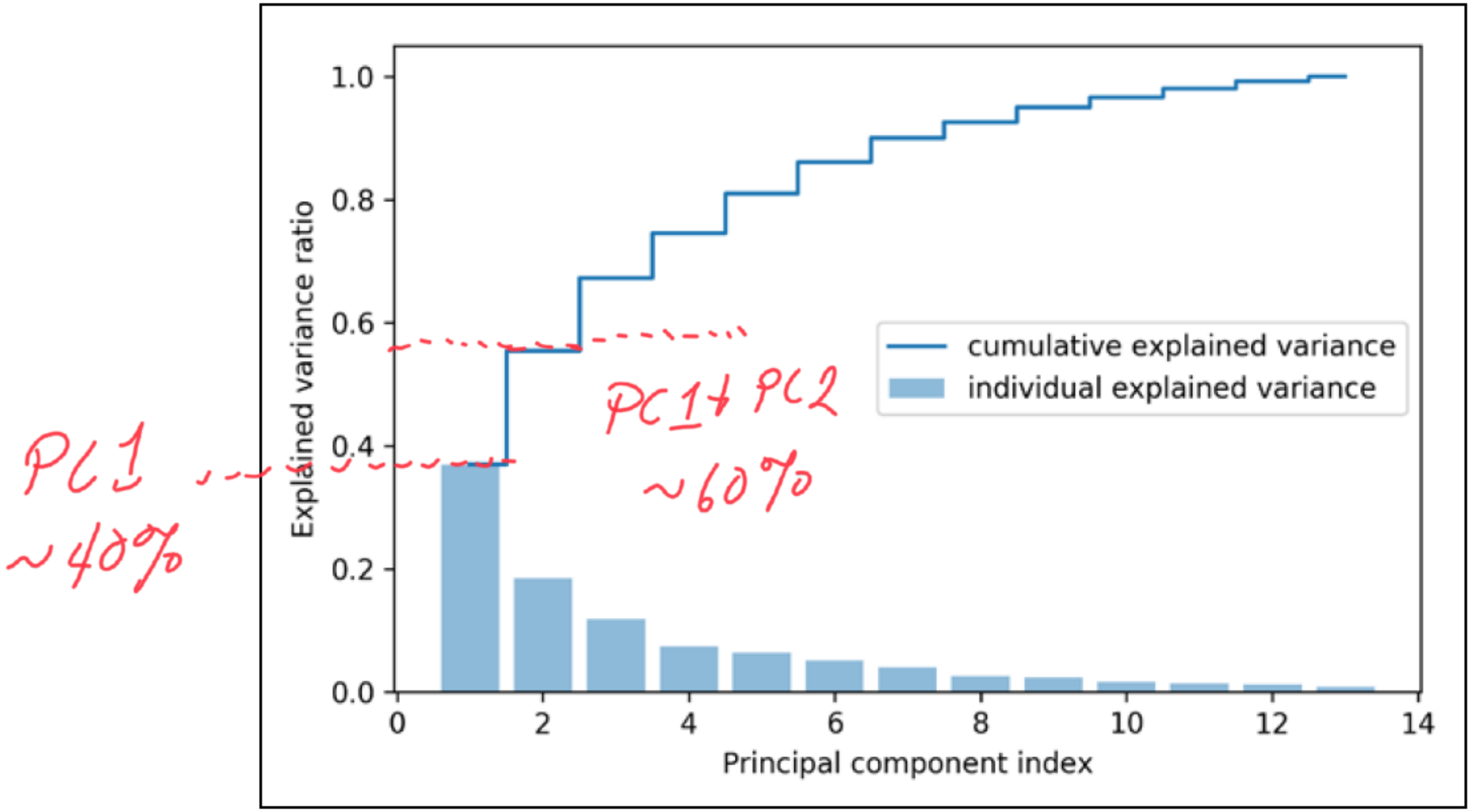
- This is a highly step to not just perform in the code…take time to analyze it as well!
- **The importance**: you will visually see the impact of the features in your dataset
- What's going on for this step? Checking eigenvalues to the aggregated sum…

$$\frac{\lambda_j}{\sum_{j=1}^{d} \lambda_j}$$

4. Sort the eigenvalues by decreasing order to rank the corresponding eigenvectors.

```
tot = sum(eigen_vals)
var_exp = [(i / tot) for i in sorted(eigen_vals,
                                   ... reverse=True)]
cum_var_exp = np.cumsum(var_exp)

# plot code of the next slide
import matplotlib.pyplot as plt
plt.bar(range(1,14), var_exp, alpha=0.5, align='center’,
                     ... label='individual explained variance')
plt.step(range(1,14), cum_var_exp, where='mid’,
                ... label='cumulative explained variance')
plt.ylabel('Explained variance ratio')
plt.xlabel('Principal component index')
plt.legend(loc='best')
plt.show()
```

4. Sort the eigenvalues by decreasing order to rank the corresponding eigenvectors.



5. Select *k* eigenvectors which correspond to the *k* largest eigenvalues, where *k* is the dimensionality of the new feature subspace ($k \leq d$).

- Create pairs to sort the eigenvector by the eigenvalues
- First Element: Eigenvalue
- 2nd Element: The associated eigenvector
- (<<eigenvalue object>>, <<eigenvector object>>)

5. Select *k* eigenvectors which correspond to the *k* largest eigenvalues, where *k* is the dimensionality of the new feature subspace ($k \leq d$).

```
eigen_pairs = [(np.abs(eigen_vals[i]), eigen_vecs[:, i])

for i in range(len(eigen_vals))]

# Sort the (eigenvalue, eigenvector) tuples from high to low
eigen_pairs.sort(key=lambda k: k[0], reverse=True)
```

6. Construct a projection matrix **W** from the "top" *k*-eigenvectors.

- Most practitioners pick the top 2 eigenvectors
- Mainly: See if you can get away the simplest amount of PC units
- There's a "*high*" chance that your top 2 can cover > 50% of features

6. Construct a projection matrix **W** from the "top" *k*-eigenvectors.

```
w = np.hstack((eigen_pairs[0][1][:, np.newaxis],
                        ... eigen_pairs[1][1][:, np.newaxis]))
print('Matrix W:\n', w)
```

pair idx

index of the pair

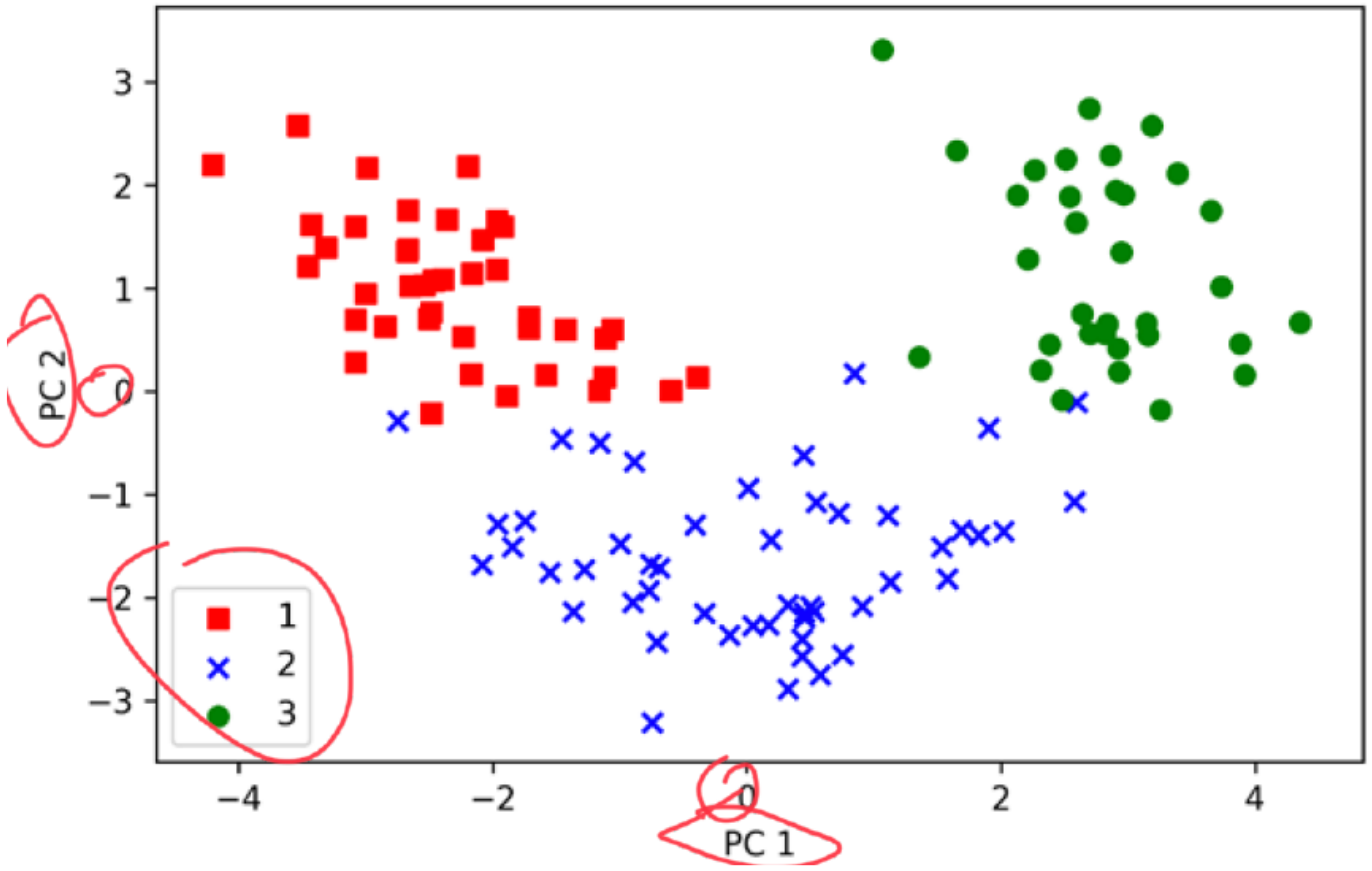
all rows for new column

```
Matrix W:
[[-0.13724218 0.50303478]
[ 0.24724326 0.16487119]
[-0.02545159 0.24456476]
[ 0.20694508 -0.11352904]
[-0.15436582 0.28974518]
[-0.39376952 0.05080104]
[-0.41735106 -0.02287338]
[ 0.30572896 0.09048885]
[-0.30668347 0.00835233]
[ 0.07554066 0.54977581]
[-0.32613263 -0.20716433]
[-0.36861022 -0.24902536]
[-0.29669651 0.38022942]]
```

7. Transform the d-dimensional input dataset **X** using the projection matrix **W** to obtain the new *k*-dimensional feature subspace.

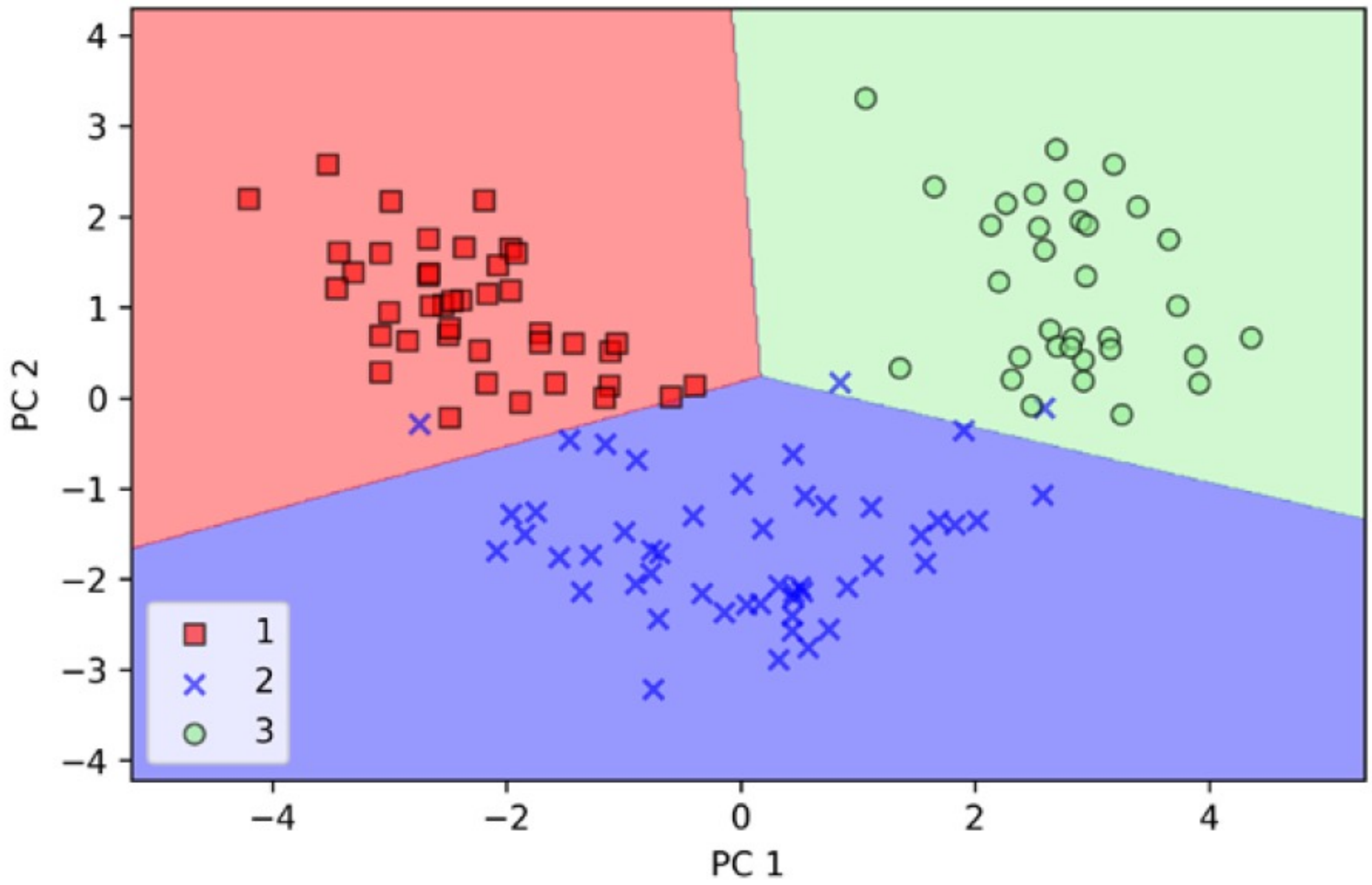
- Transformation: $X' = XW$

```
X_train_pca = X_train_std.dot(w)
colors = ['r', 'b', 'g']
markers = ['s', 'x', 'o']
for l, c, m in zip(np.unique(y_train), colors, markers):
     ... plt.scatter(X_train_pca[y_train==l, 0],
     ... X_train_pca[y_train==l, 1],
     ... c=c, label=l, marker=m)
plt.xlabel('PC 1')
plt.ylabel('PC 2')
plt.legend(loc='lower left')
plt.show()
```

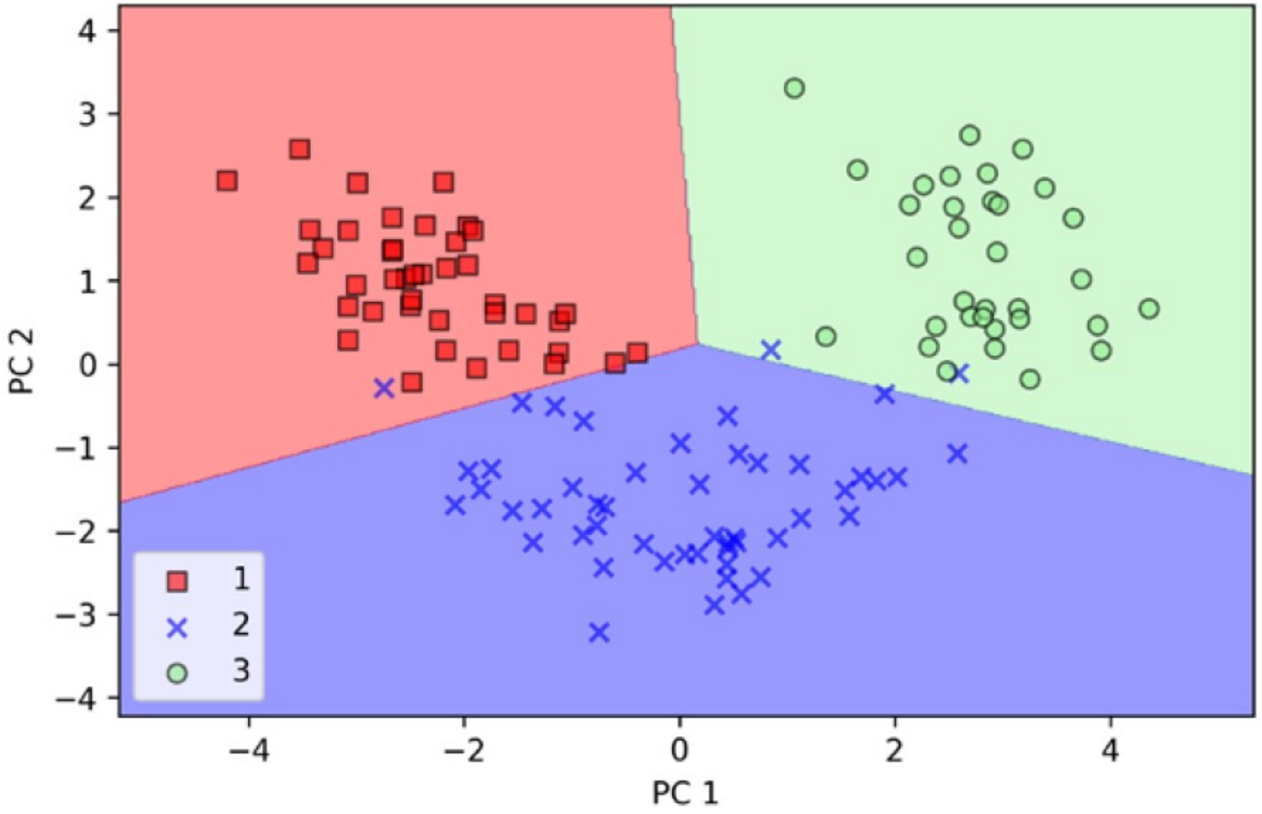

PC 2
PC 1
1
2
3

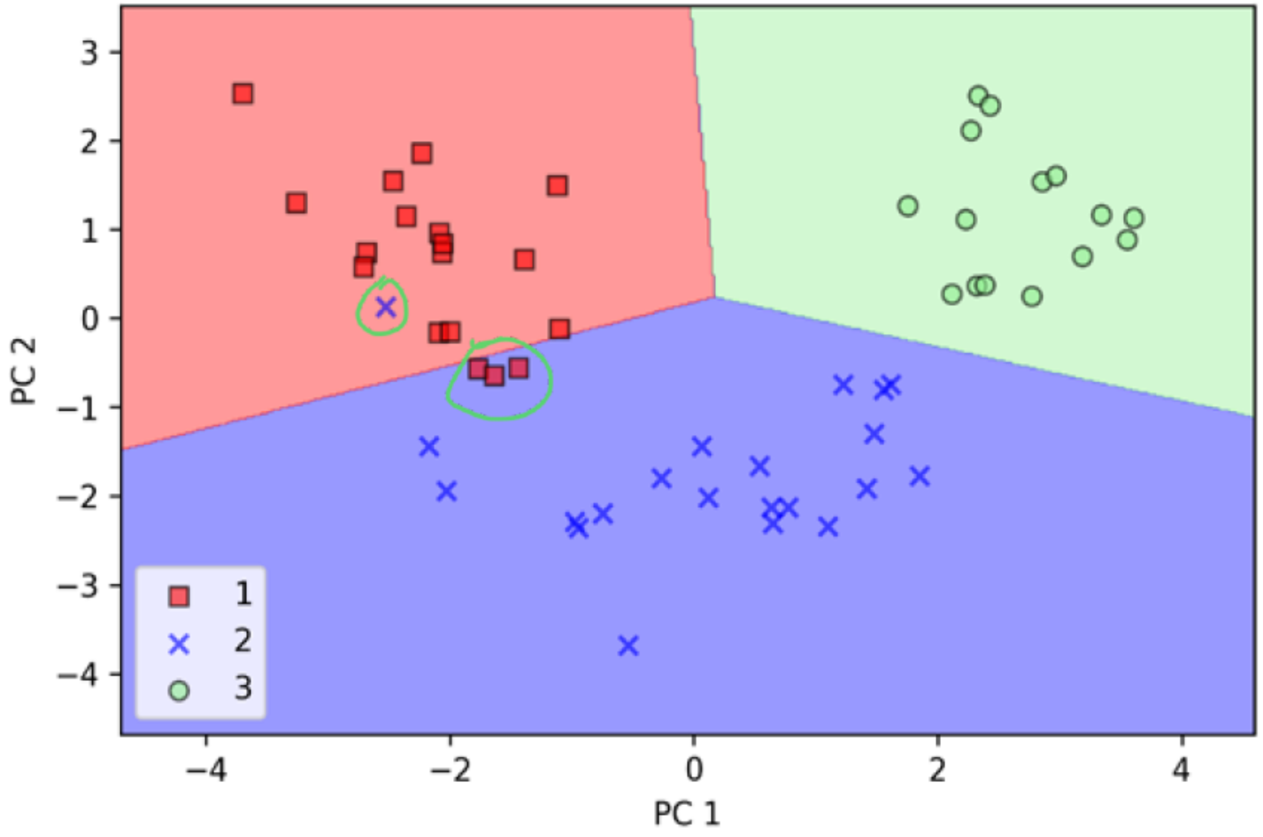
The scikit-learn implementation using Logistic Regression:

```
from sklearn.linear_model import LogisticRegression
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
lr = LogisticRegression()
X_train_pca = pca.fit_transform(X_train_std)
X_test_pca = pca.transform(X_test_std)
lr.fit(X_train_pca, y_train) #assuming you did get y_train @ this point as well
plot_decision_regions(X_train_pca, y_train, classifier=lr)
plt.xlabel('PC 1')
plt.ylabel('PC 2')
plt.legend(loc='lower left')
plt.show()

# did not show the fancy coloring code for the output
# see textbook for that part.
```



PC 2
PC 1
4
3
2
1
0
−1
−2
−3
−4
−4
−2
0
2
4
1
2
3





```
plot_decision_regions(X_test_pca, y_test, classifier=lr)
```

LDA

- In the context of feature extraction:
  - Helps not to overfit the model
  - **This is a supervised method**

- **LDAs** are widely used when you have a common problem: **too many features**!!!

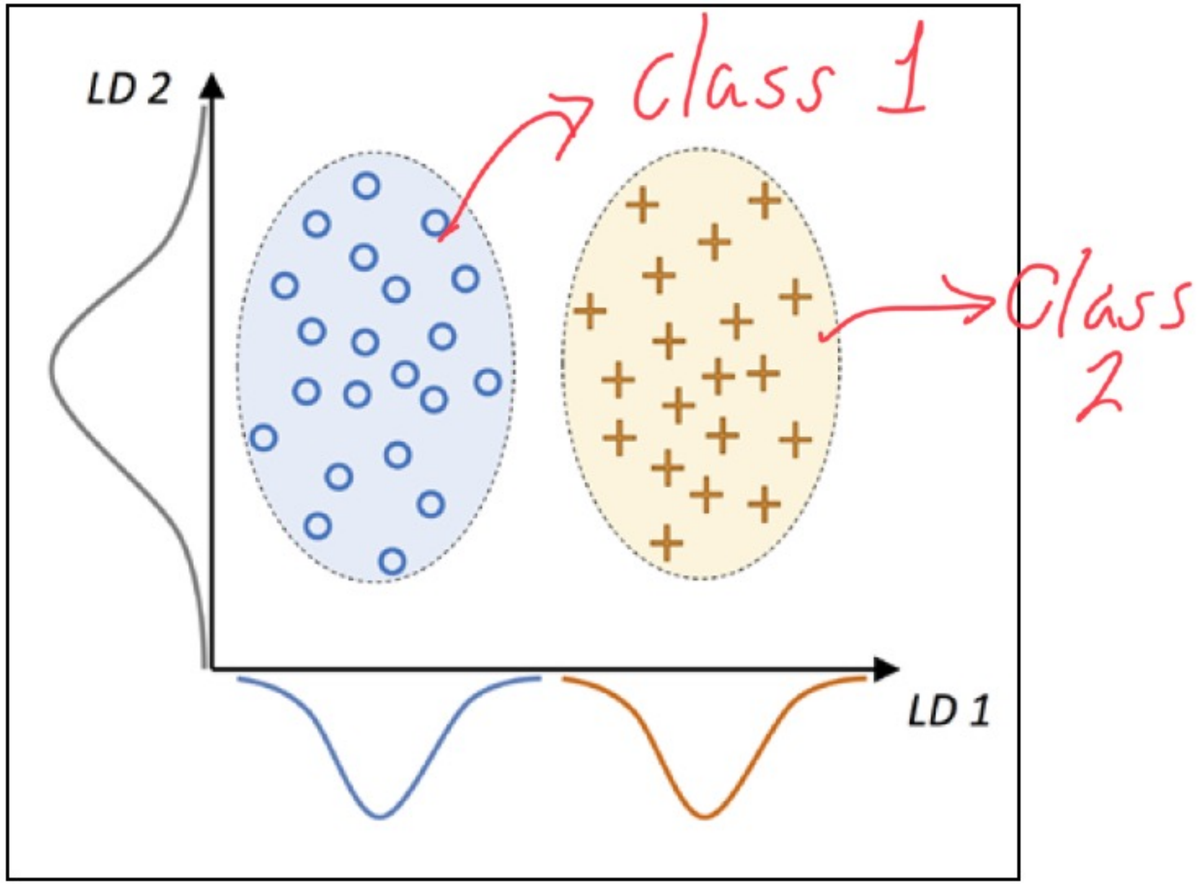
- This will force you to perform a dimensional reduction without losing the information of all the dataset using its class labels and

- **The big assumptions:**

- The data is normally distributed

- Features are independent from each other

- Covariance matrices for all classes are identical



7-STEPS to LDA

1. Standardize the *d*-dimensional dataset (*d is the number of features*).

2. For each class, compute the *d*-dimensional mean vector.

3. Construct the between-class scatter matrix $\boldsymbol{S_B}$ and the within-class scatter matrix $\boldsymbol{S_W}$.

4. Compute the eigenvectors and corresponding eigenvalues of the matrix $\boldsymbol{S_W^{-1} S_B}$

5. Sort the eigenvalues by decreasing order to rank the corresponding eigenvectors.

6. Choose the *k* eigenvectors that correspond to the *k* largest eigenvalues to construct a *d* x *k*-dimensional transformation matrix **W**; the eigenvectors are the columns of this matrix.

7. Project the samples onto the new feature subspace using the transformation matrix **W**.

1. Standardize the *d*-dimensional dataset (*d is the number of features*).

2. For each class, compute the *d*-dimensional mean vector.

$$\boldsymbol{m}_i = \frac{1}{n_i}\sum_{x\in D_i}^{c} \boldsymbol{x}_m \Rightarrow \boldsymbol{m}_i = \begin{bmatrix} \mu_{i,alcohol} \\ \mu_{i,malic\ acid} \\ \vdots \\ \mu_{i,proline} \end{bmatrix} \quad i\in\{1,2,3\}$$

$$m_i = \begin{bmatrix} \mu_{i,ft.1} \\ \mu_{i,ft.2} \\ \vdots \\ \mu_{i,ft\alpha} \end{bmatrix} \quad i \in \{1 : \# \text{ of classes}\}$$

2. For each class, compute the *d*-dimensional mean vector.

```
np.set_printoptions(precision=4)
mean_vecs = []
for label in range(1,4):
mean_vecs.append(np.mean(
     ... X_train_std[y_train==label], axis=0))
     ... print('MV %s: %s\n' %(label, mean_vecs[label-1]))
print('MV %s: %s\n' %(label, mean_vecs[label-1]))
```

```
MV 1: [ 0.9066 -0.3497 0.3201 -0.7189 0.5056 0.8807 0.9589 -0.5516
     0.5416 0.2338 0.5897 0.6563 1.2075]
MV 2: [-0.8749 -0.2848 -0.3735 0.3157 -0.3848 -0.0433 0.0635 -     0.0946
0.0703 -0.8286 0.3144 0.3608 -0.7253]
MV 3: [ 0.1992 0.866 0.1682 0.4148 -0.0451 -1.0286 -1.2876 0.8287
     -0.7795 0.9649 -1.209 -1.3622 -0.4013]
```

3. Construct the between-class scatter matrix $\boldsymbol{S_B}$ and the within-class scatter matrix $\boldsymbol{S_W}$.

- The within-class scatter matrix:

$$\boldsymbol{S}_W = \sum_{i=1}^{c} \boldsymbol{S}_i$$ $\forall$ classes

$$\boldsymbol{S}_i = \sum_{\boldsymbol{x} \in D_i}^{c} (\boldsymbol{x} - \boldsymbol{m}_i)(\boldsymbol{x} - \boldsymbol{m}_i)^T$$ For class i

3. Construct the between-class scatter matrix $\boldsymbol{S_B}$ and the within-class scatter matrix $\boldsymbol{S_W}$.

- The within-class scatter matrix:

```
d = 13 # number of features
S_W = np.zeros((d, d))
for label, mv in zip(range(1, 4), mean_vecs):
      ... class_scatter = np.zeros((d, d))
for row in X_train_std[y_train == label]:
      ... row, mv = row.reshape(d, 1), mv.reshape(d, 1)
      ... class_scatter += (row - mv).dot((row - mv).T)
      ... S_W += class_scatter
print('Class label distribution: %s' % np.bincount(y_train)[1:])

Class label distribution: [41 50 33]… distribution is unbalanced
```

3. Construct the between-class scatter matrix S_B and the within-class scatter matrix S_W .

- The between-class scatter matrix:

$$S_B = \sum_{i=1}^{c} n_i (\boldsymbol{m}_i - \boldsymbol{m})(\boldsymbol{m}_i - \boldsymbol{m})^T$$

- ***m***: the overall average that includes samples from other classes

3. Construct the between-class scatter matrix S_B and the within-class scatter matrix S_W .

- The between-class scatter matrix:

```
mean_overall = np.mean(X_train_std, axis=0)
d = 13 # number of features
S_B = np.zeros((d, d))
for i, mean_vec in enumerate(mean_vecs):
     ... n = X_train[y_train == i + 1, :].shape[0]
     ... mean_vec = mean_vec.reshape(d, 1) # make column vector
     ... mean_overall = mean_overall.reshape(d, 1)
     ... S_B += n * (mean_vec - mean_overall).dot(
     ... (mean_vec - mean_overall).T)
```

4. Compute the eigenvectors and corresponding eigenvalues of the matrix $\boldsymbol{S_W^{-1} S_B}$

```
eigen_vals, eigen_vecs =\
... np.linalg.eig(np.linalg.inv(S_W).dot(S_B))
```

5. Sort the eigenvalues by decreasing order to rank the corresponding eigenvectors.

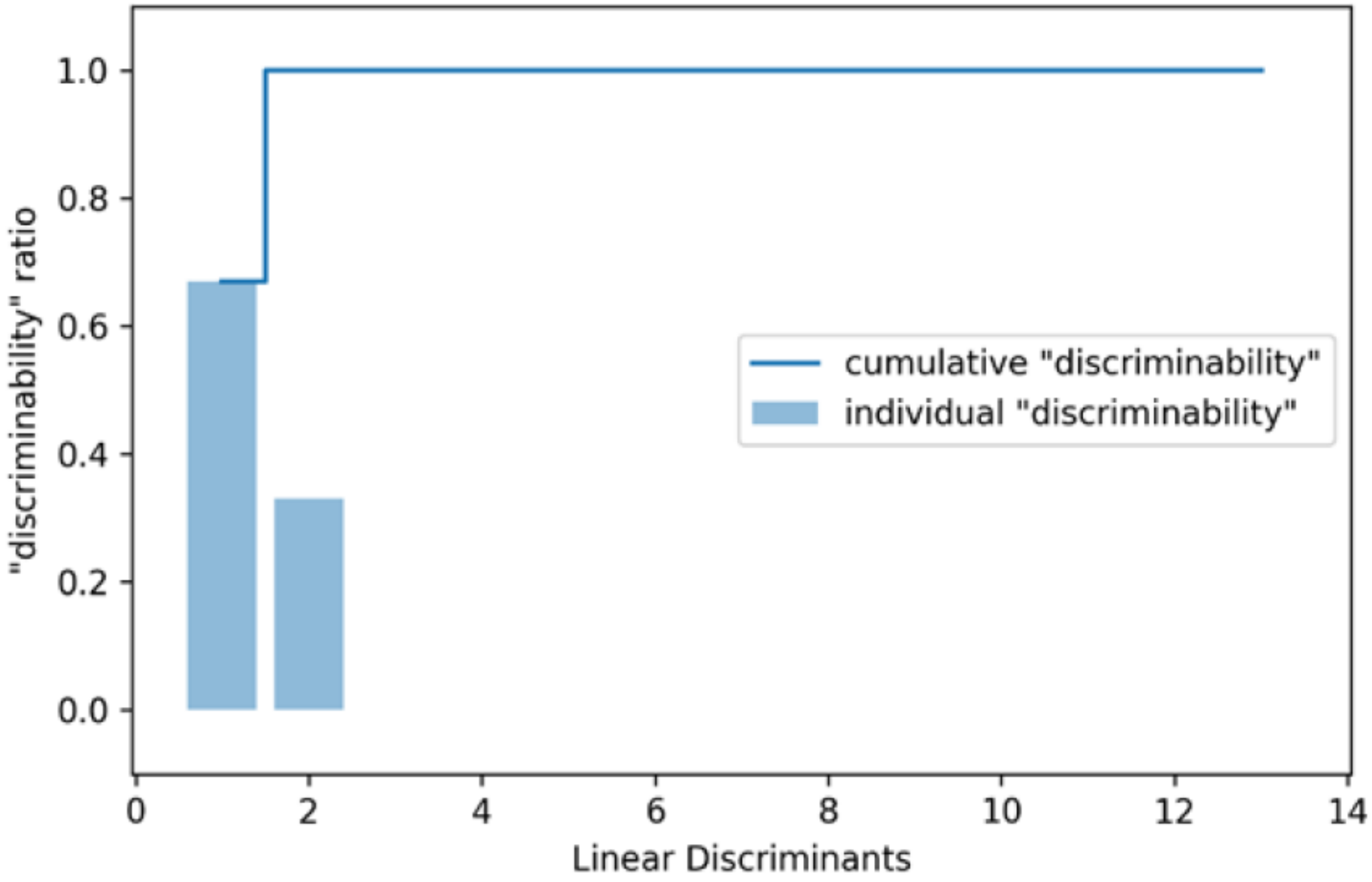
```
eigen_pairs = [(np.abs(eigen_vals[i]), eigen_vecs[:,i])
     ... for i in range(len(eigen_vals))]

eigen_pairs = sorted(eigen_pairs,
     ... key=lambda k: k[0], reverse=True)
```

```
print('Eigenvalues in descending order:\n')
for eigen_val in eigen_pairs:
          print(eigen_val[0])
```

```
Eigenvalues in descending order:
349.617808906
172.76152219
3.78531345125e-14
2.11739844822e-14
1.51646188942e-14
1.51646188942e-14
1.35795671405e-14
1.35795671405e-14
7.58776037165e-15
5.90603998447e-15
5.90603998447e-15
2.25644197857e-15
0.0
```

Print out the LD magnitudes

```
tot = sum(eigen_vals.real)
discr = [(i / tot) for i in sorted(eigen_vals.real,
reverse=True)]
cum_discr = np.cumsum(discr)

plt.bar(range(1, 14), discr, alpha=0.5, align='center',
... label='individual "discriminability"')
plt.step(range(1, 14), cum_discr, where='mid',
... label='cumulative "discriminability"')
plt.ylabel('"discriminability" ratio')
plt.xlabel('Linear Discriminants')
plt.ylim([-0.1, 1.1])
plt.legend(loc='best')
plt.show()
```



"discriminability" ratio
Linear Discriminants
cumulative "discriminability"
individual "discriminability"
0.0
0.2
0.4
0.6
0.8
1.0
0
2
4
6
8
10
12
14

6. Choose the *k* eigenvectors that correspond to the *k* largest eigenvalues to construct a *d x k*-dimensional transformation matrix **W**; the eigenvectors are the columns of this matrix.

- The first two are the main LDs

```
w = np.hstack((eigen_pairs[0][1][:, np.newaxis].real,
          ... eigen_pairs[1][1][:, np.newaxis].real))
```

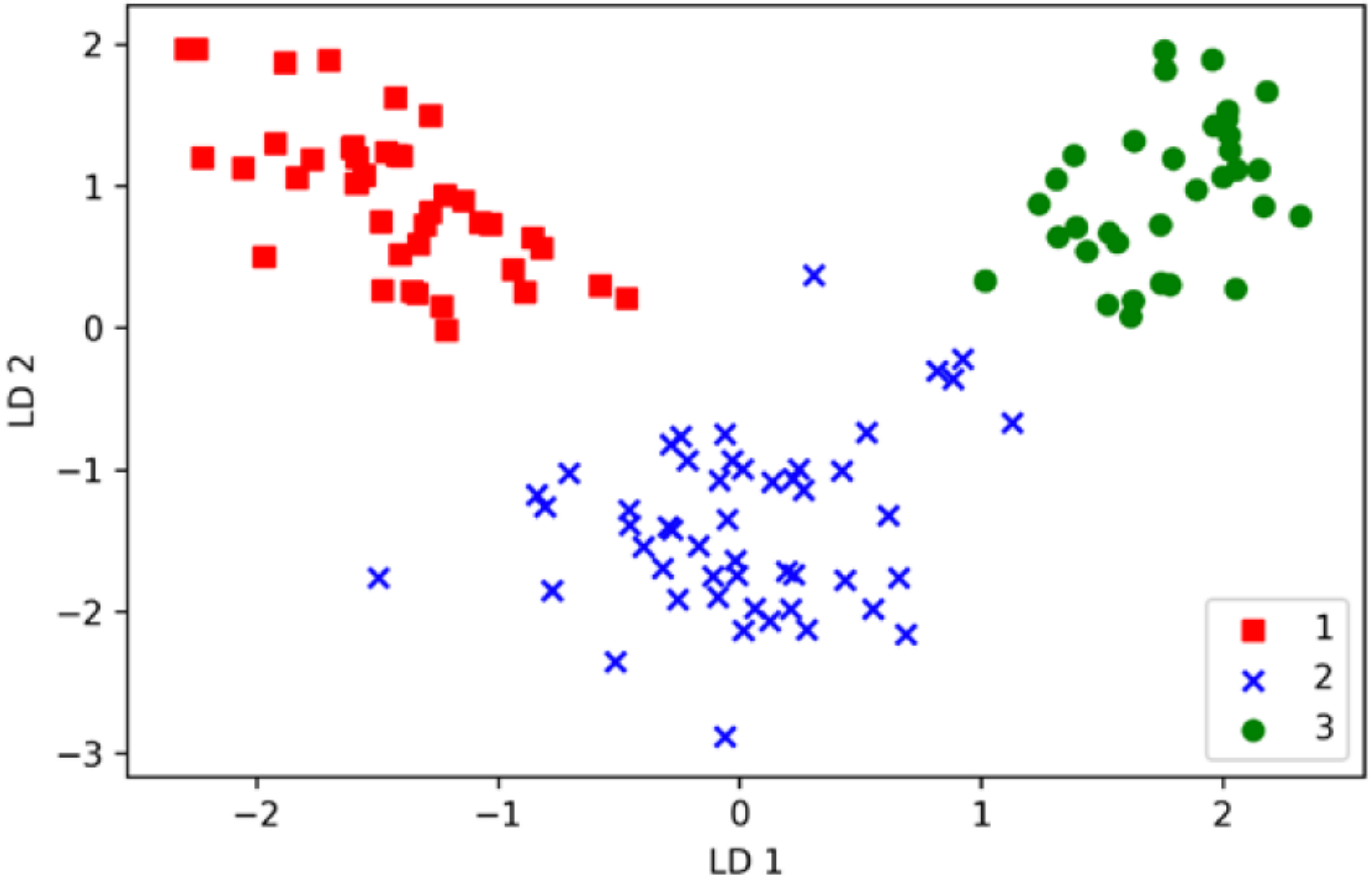
```
print('Matrix W:\n', w)
```

```
Matrix W:
[[-0.1481 -0.4092]
[ 0.0908 -0.1577]
[-0.0168 -0.3537]
[ 0.1484 0.3223]
[-0.0163 -0.0817]
[ 0.1913 0.0842]
[-0.7338 0.2823]
[-0.075 -0.0102]
[ 0.0018 0.0907]
[ 0.294 -0.2152]
[-0.0328 0.2747]
[-0.3547 -0.0124]
[-0.3915 -0.5958]]
```

7. Project the samples onto the new feature subspace using the transformation matrix **W**.

- The projection to new subspace: $X' = XW$

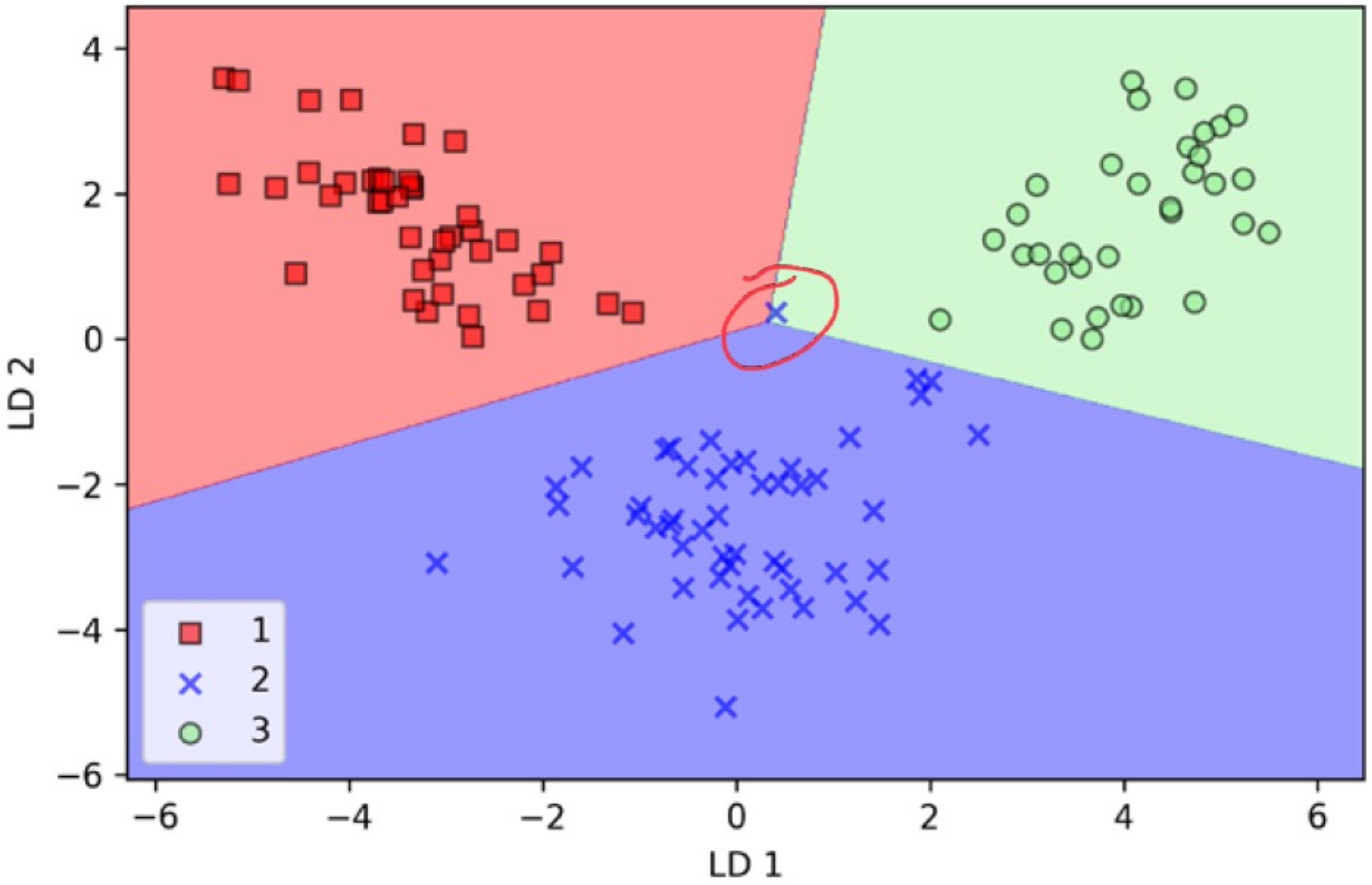
```
X_train_lda = X_train_std.dot(w)
colors = ['r', 'b', 'g']
markers = ['s', 'x', 'o']
for l, c, m in zip(np.unique(y_train), colors, markers):
     ... plt.scatter(X_train_lda[y_train==l, 0],
     ... X_train_lda[y_train==l, 1] * (-1),
     ... c=c, label=l, marker=m)
plt.xlabel('LD 1')
plt.ylabel('LD 2')
plt.legend(loc='lower right')
plt.show()
```



LD 2
LD 1
2
1
0
−1
−2
−3
−2
−1
0
1
2
1
2
3

The scikit-version:

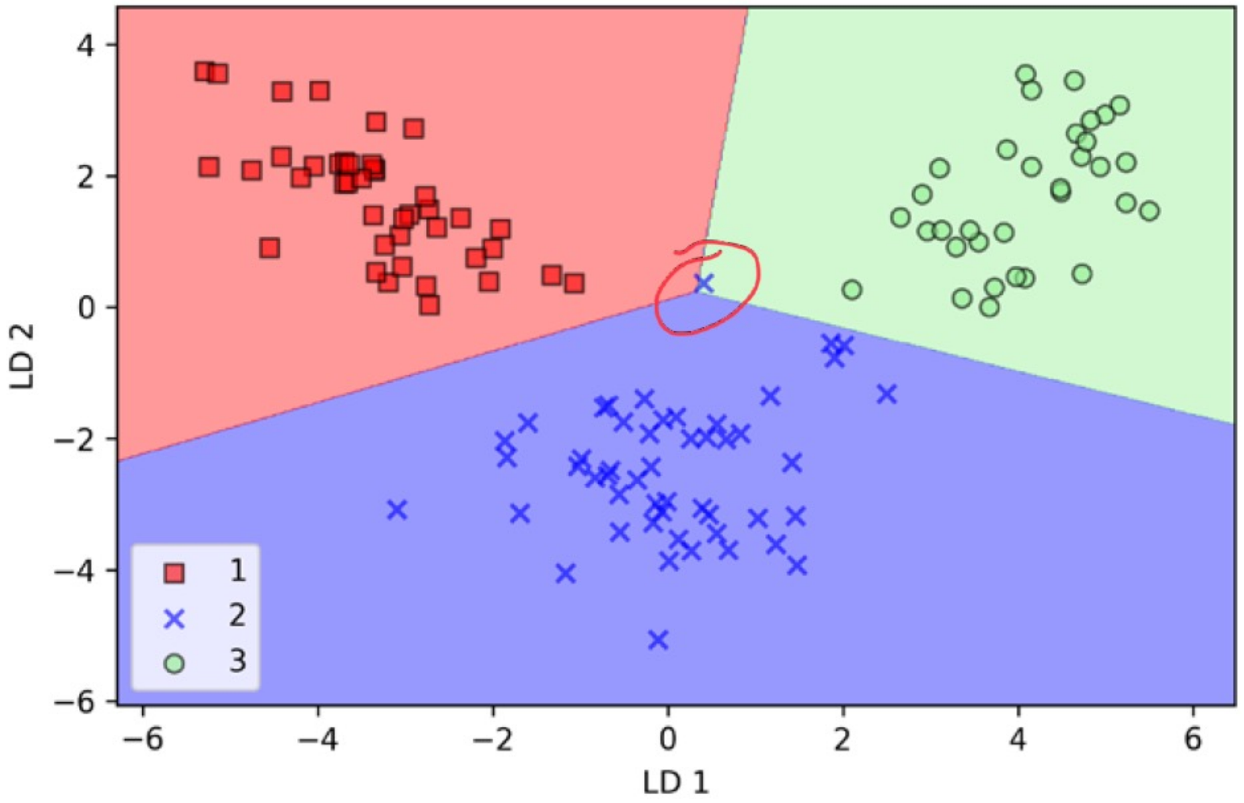
```
lr = LogisticRegression()
lr = lr.fit(X_train_lda, y_train)
plot_decision_regions(X_train_lda, y_train,
        ... classifier=lr)

plt.xlabel('LD 1')
plt.ylabel('LD 2')
plt.legend(loc='lower left')
plt.show()
```



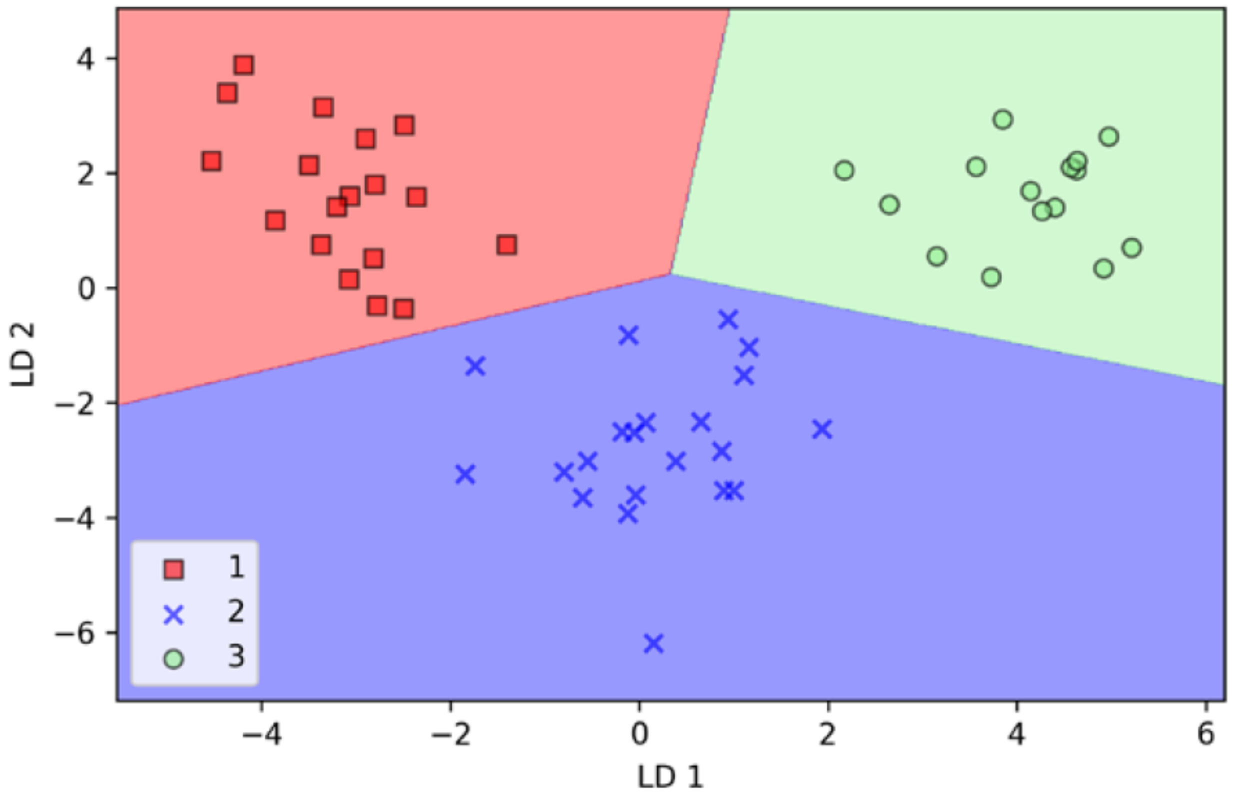
4
2
0
−2
−4
−6
LD 2
−6
−4
−2
0
2
4
6
LD 1
1
2
3

The test split data:

```
X_test_lda = lda.transform(X_test_std)
plot_decision_regions(X_test_lda, y_test,
             ... classifier=lr)

plt.xlabel('LD 1')
plt.ylabel('LD 2')
plt.legend(loc='lower left')
plt.show()
```



LD 2
LD 1
1
2
3



LD 2
LD 1
1
2
3

Kernel PCA

- Kernel to the rescue for **non-linear problems** *(higher dimension)*

- The transformation for KPCA is the key:

$$\boxed{\phi : \mathbb{R}^d \rightarrow \mathbb{R}^k \quad (k >> d)}$$

$$\boldsymbol{x} = [x_1,\ x_2]^T \Longrightarrow x \in \mathbb{R}^2$$

$$\downarrow \phi$$

$$\mathbf{z} = \left[x_1^2, \sqrt{2x_1x_2}, x_2^2\right]^T \Longrightarrow \text{This can be different!}$$

- **Step 1:** Dimensional Transformation

- **Step 2:** Do all the PCA steps!

  - Covariance (*review*):

$$\sigma_{jk} = \frac{1}{n}\sum_{i=1}^{n}\left(x_j^{(i)} - \mu_j\right)\left(x_k^{(i)} - \mu_k\right)$$

  - Covariance Matrix (*review*):

$$\Sigma = \frac{1}{n}\sum_{i=1}^{n}\boldsymbol{x}^{(i)}\,\boldsymbol{x}^{(i)^T}$$

- **Step 3:** KPCA Covariance Matrix:

$$\Sigma = \frac{1}{n}\sum_{i=1}^{n}\phi\left(\boldsymbol{x}^{(i)}\right)\phi(\boldsymbol{x}^{(i)})^{T}$$

- Eigenvectors:

$$\Sigma \boldsymbol{v} = \lambda \boldsymbol{v}$$

- Eigenvectors:

$$\Sigma \boldsymbol{v} = \lambda \boldsymbol{v}$$

$$\Rightarrow \frac{1}{n}\sum_{i=1}^{n} \phi\left(\boldsymbol{x}^{(i)}\right)\phi\left(\boldsymbol{x}^{(i)}\right)^{T} \boldsymbol{v} = \lambda \boldsymbol{v}$$

$$\Rightarrow v = \frac{1}{n\lambda}\sum_{i=1}^{n} \phi\left(\boldsymbol{x}^{(i)}\right)\phi\left(\boldsymbol{x}^{(i)}\right)^{T} \boldsymbol{v}$$

$$= \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{a}^{(i)} \phi\left(\boldsymbol{x}^{(i)}\right)$$

$$a^{(i)} = \frac{\phi(x^{(i)})^T}{\lambda}$$

- **Kernel Function**:

$$\kappa\left(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)}\right)=\phi\left(\boldsymbol{x}^{(i)}\right)^{T}\phi\left(\boldsymbol{x}^{(j)}\right)$$

- This is more useful than trying to figure out the Eigen pairs

- The kernel PCA are samples are already projected (transformed) to the PC components

- There are different flavors of kernel functions

- **Polynomial Kernel**:

$$\kappa\left(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)}\right)=\left(\boldsymbol{x}^{(i)T}\boldsymbol{x}^{(j)}+\theta\right)^{p}$$

- **Hyperbolic Tangent (sigmoid) kernel**:

$$\kappa\left(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)}\right)=\tanh\left(\eta\boldsymbol{x}^{(i)T}\boldsymbol{x}^{(j)}+\theta\right)$$

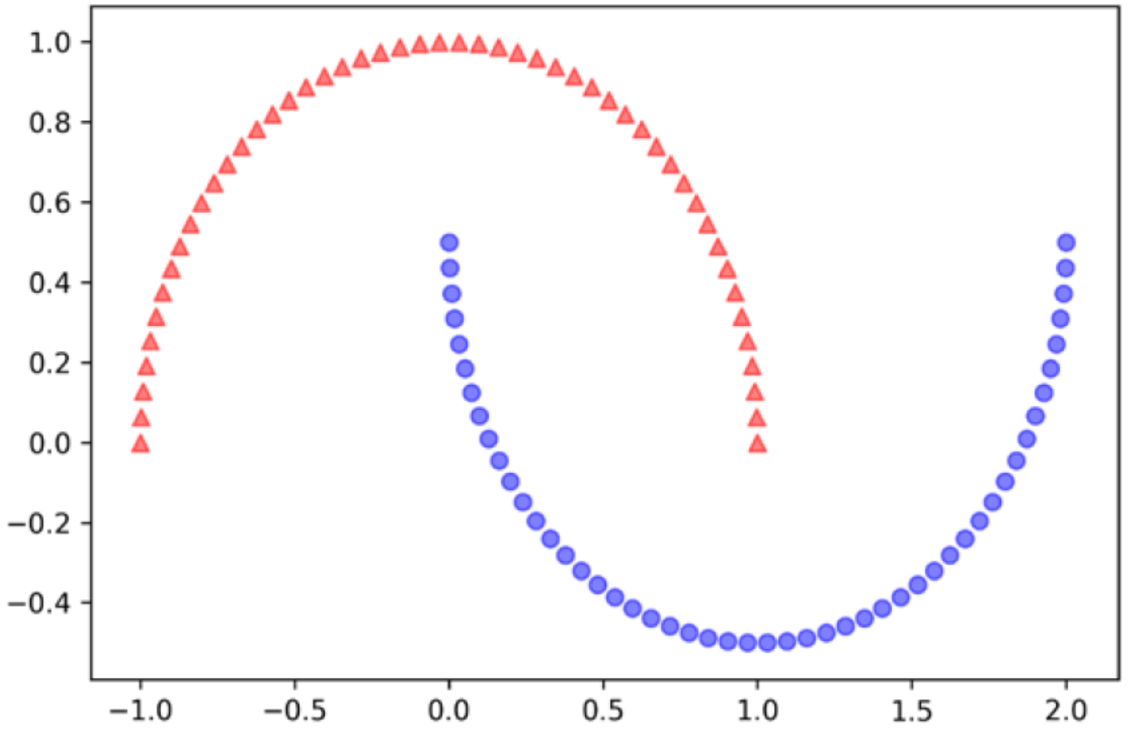
- **Radial Basis Function (Gaussian) kernel**:

$$\kappa\left(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)}\right)=\exp\left(-\frac{\left\|\boldsymbol{x}^{(i)}-\boldsymbol{x}^{(j)}\right\|^{2}}{2\sigma^{2}}\right) \longrightarrow \boxed{\gamma=\frac{1}{2\sigma}}$$

- Half Moon Example:

- Authors: 2 Scenarios (homebrew vs. PCA)

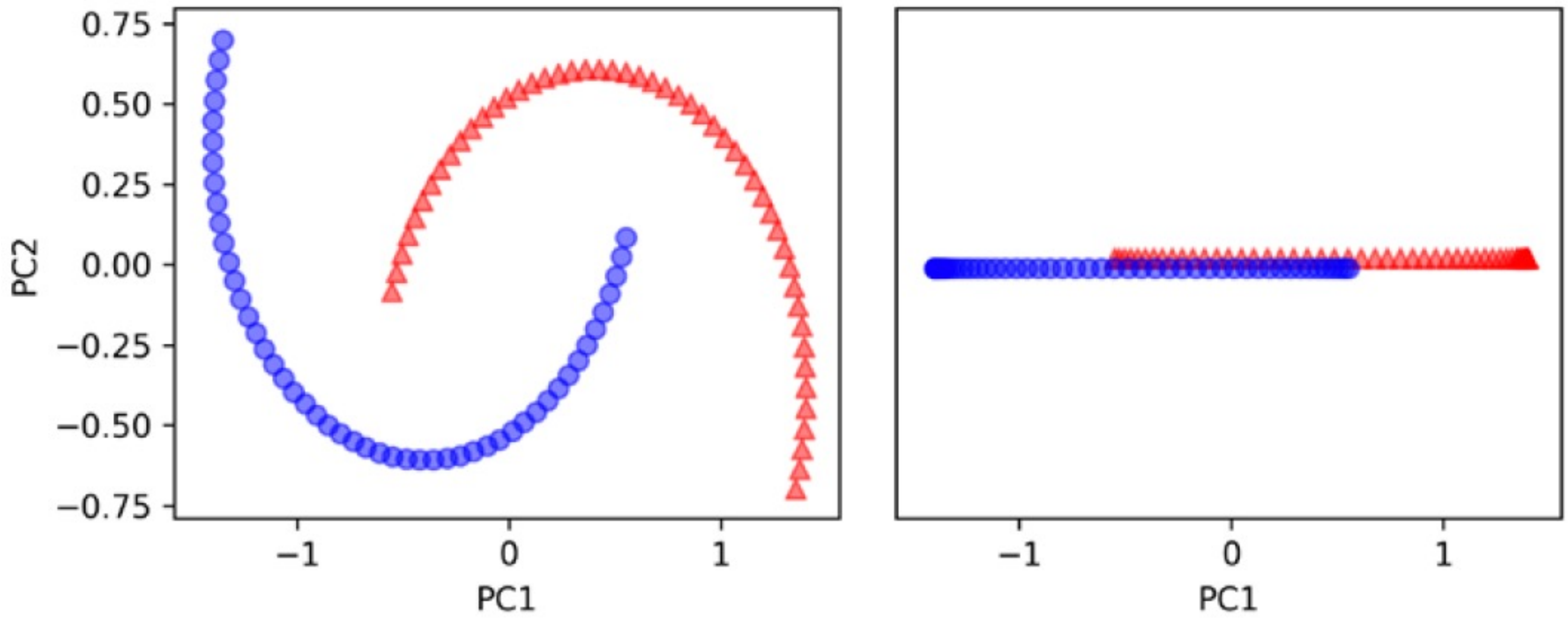
```
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=100,
              ... random_state=123)

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

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

plt.show()
```



- **Normal** PCA

```
from sklearn.decomposition import PCA
scikit_pca = PCA(n_components=2)
X_spca = scikit_pca.fit_transform(X)
fig, ax = plt.subplots(nrows=1,ncols=2, figsize=(7,3))
ax[0].scatter(X_spca[y==0, 0], X_spca[y==0, 1],
        ... color='red', marker='^', alpha=0.5)
ax[0].scatter(X_spca[y==1, 0], X_spca[y==1, 1],
        ... color='blue', marker='o', alpha=0.5)
ax[1].scatter(X_spca[y==0, 0], np.zeros((50,1))+0.02,
        ... color='red', marker='^', alpha=0.5)
ax[1].scatter(X_spca[y==1, 0], np.zeros((50,1))-0.02,
        ... color='blue', marker='o', alpha=0.5)
ax[0].set_xlabel('PC1')
ax[0].set_ylabel('PC2')
ax[1].set_ylim([-1, 1])
ax[1].set_yticks([])
ax[1].set_xlabel('PC1')
plt.show()
```

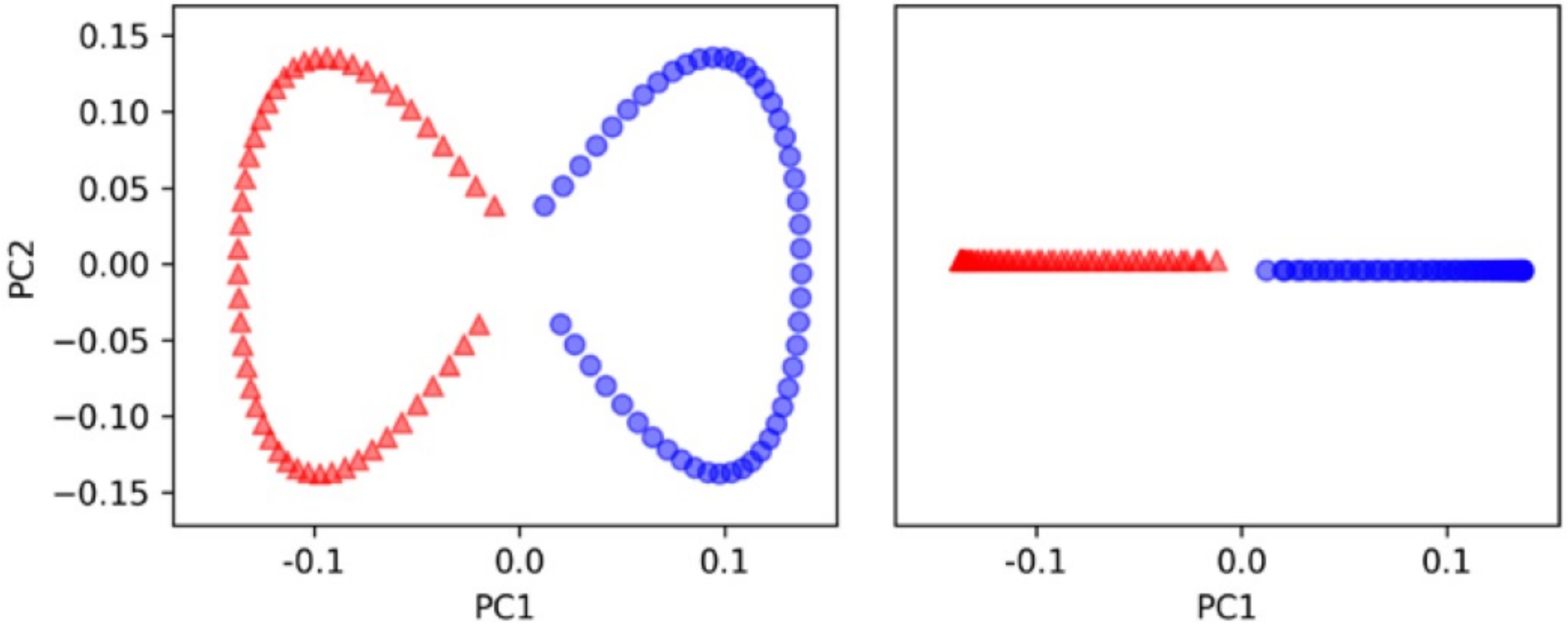
- **Normal** PCA Results



- Homebrew RBF Kernel PCA

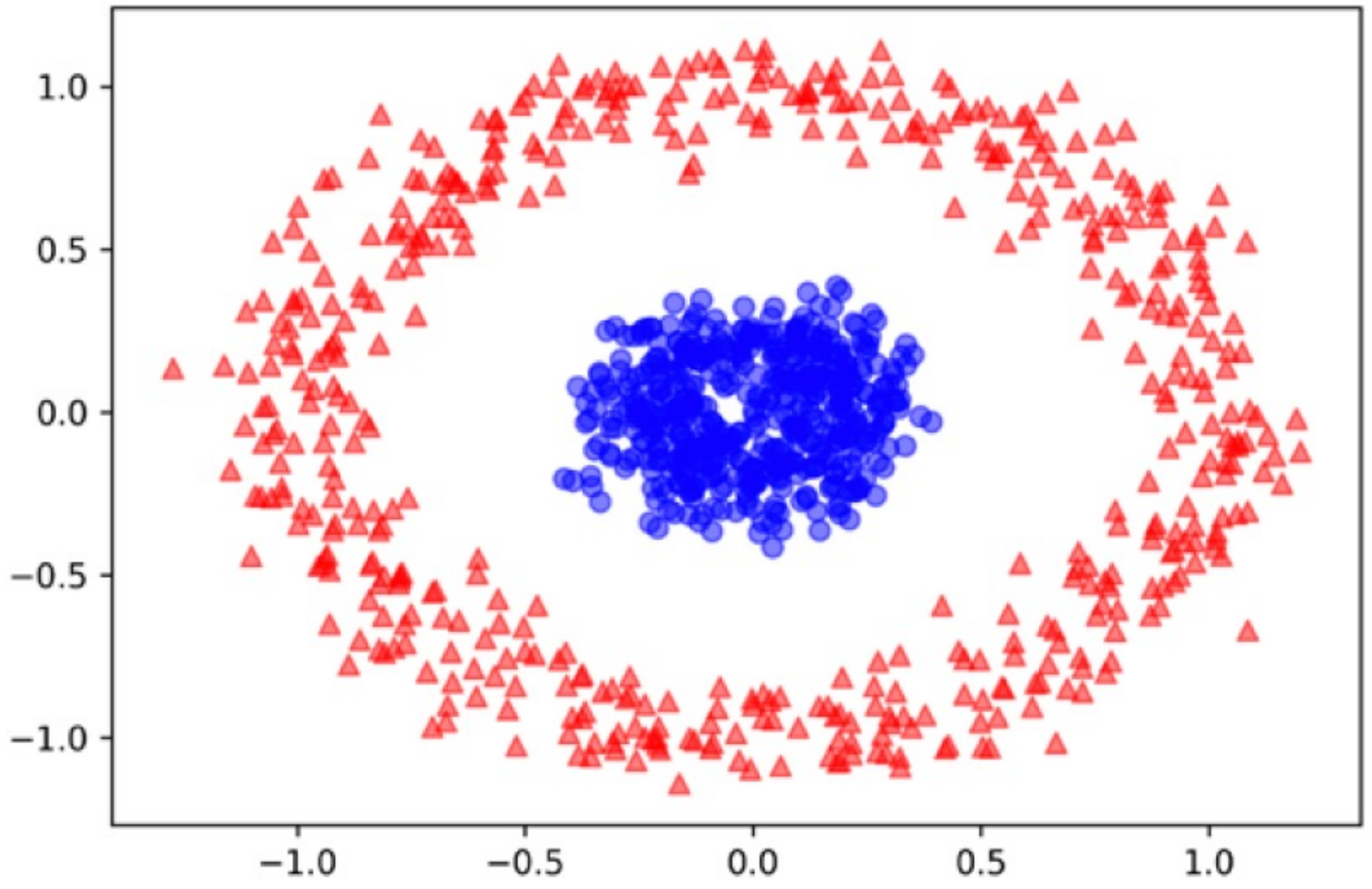
```
from sklearn.decomposition import PCA
X_kpca = rbf_kernel_pca(X, gamma=15, n_components=2)

fig, ax = plt.subplots(nrows=1,ncols=2, figsize=(7,3))
ax[0].scatter(X_kpca[y==0, 0], X_kpca[y==0, 1],
        ... color='red', marker='^', alpha=0.5)
ax[0].scatter(X_kpca[y==1, 0], X_kpca[y==1, 1],
        ... color='blue', marker='o', alpha=0.5)
ax[1].scatter(X_kpca[y==0, 0], np.zeros((50,1))+0.02,
        ... color='red', marker='^', alpha=0.5)
ax[1].scatter(X_kpca[y==1, 0], np.zeros((50,1))-0.02,
        ... color='blue', marker='o', alpha=0.5)
ax[0].set_xlabel('PC1')
ax[0].set_ylabel('PC2')
ax[1].set_ylim([-1, 1])
ax[1].set_yticks([])
ax[1].set_xlabel('PC1')
plt.show()
```

- Homebrew RBF Kernel PCA Results

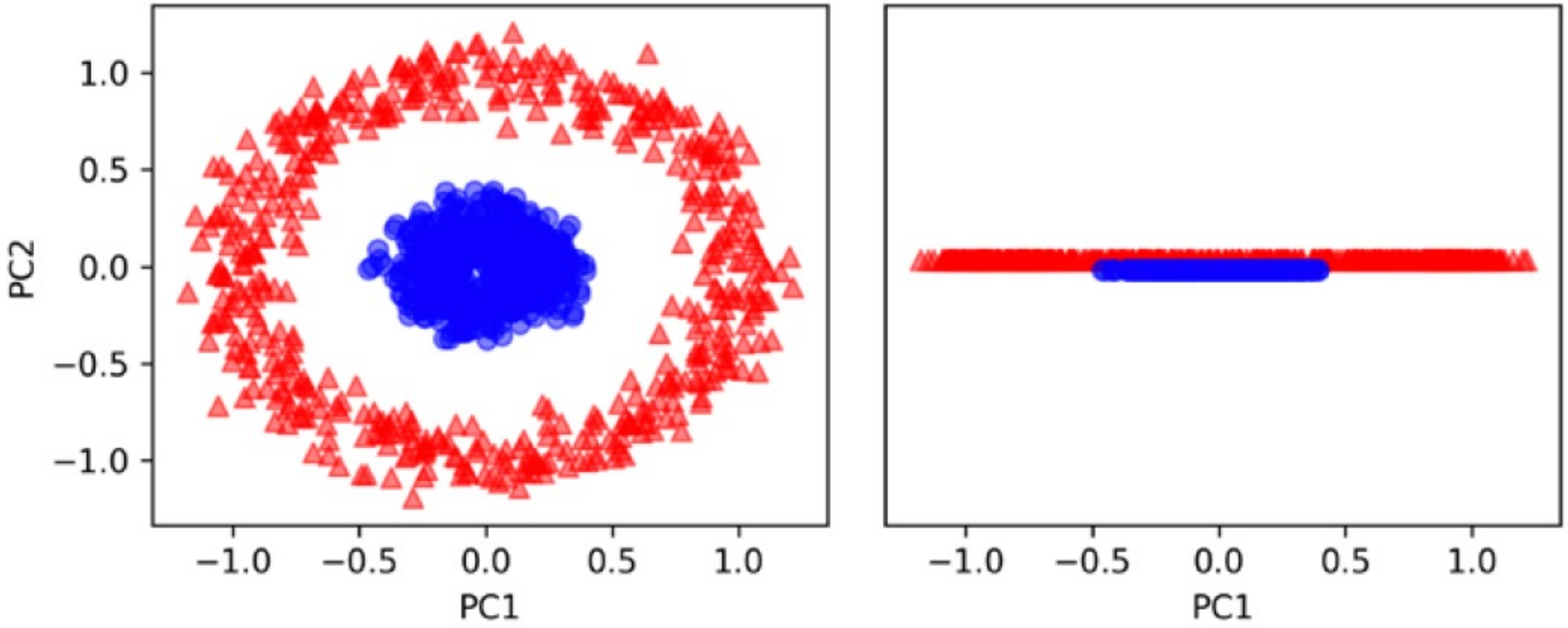


- Concentric Circles Example:

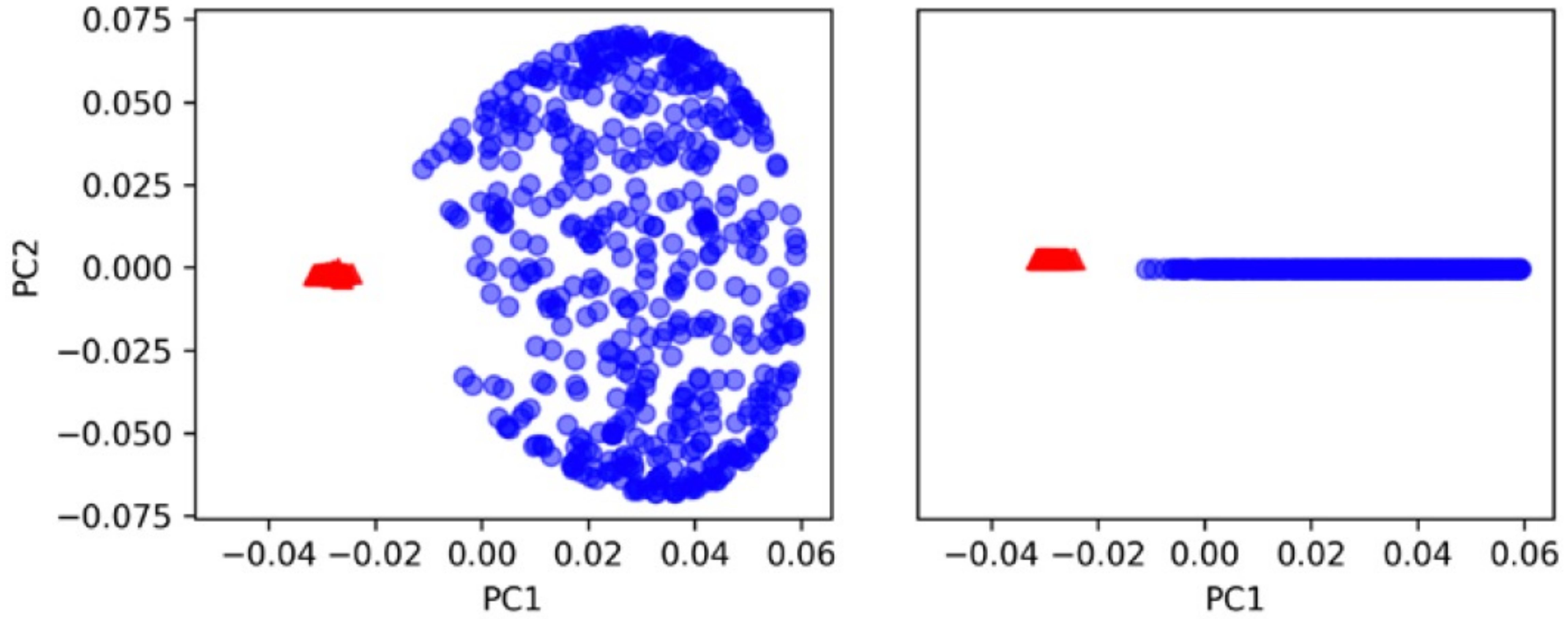
- Authors: 2 Scenarios (homebrew vs. PCA)



- Normal PCA Results



- Homebrew RBF Kernel PCA Results



Scikit-Learn Implementation

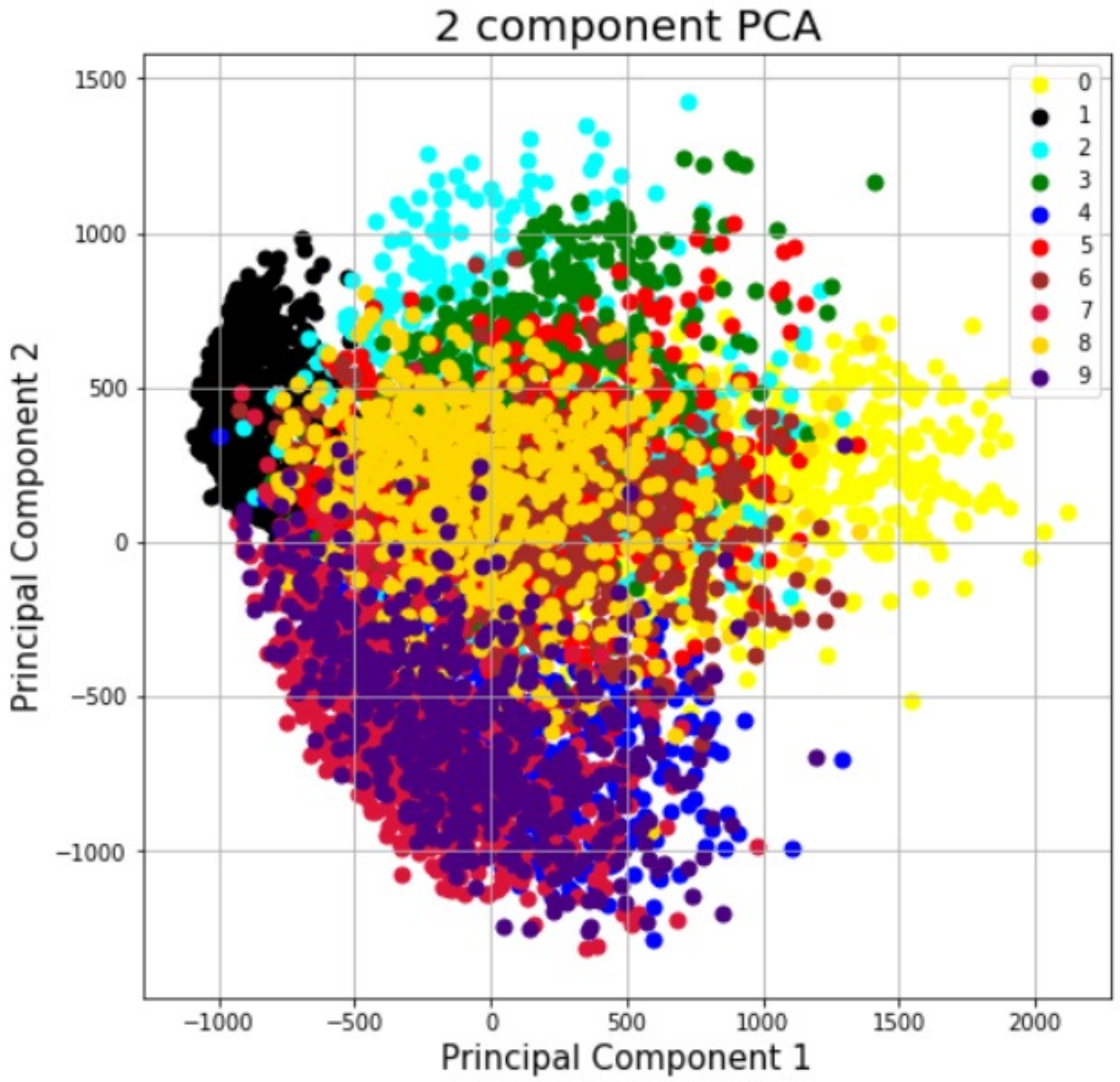
```
from sklearn.decomposition import KernelPCA
X, y = make_moons(n_samples=100, random_state=123)
scikit_kpca = KernelPCA(n_components=2,
                                  ... kernel='rbf', gamma=15)
X_skernpca = scikit_kpca.fit_transform(X)

plt.scatter(X_skernpca[y==0, 0], X_skernpca[y==0, 1],
                        ... color='red', marker='^', alpha=0.5)
plt.scatter(X_skernpca[y==1, 0], X_skernpca[y==1, 1],
                        ... color='blue', marker='o',
alpha=0.5)
plt.xlabel('PC1')
plt.ylabel('PC2’)
plt.show()
```

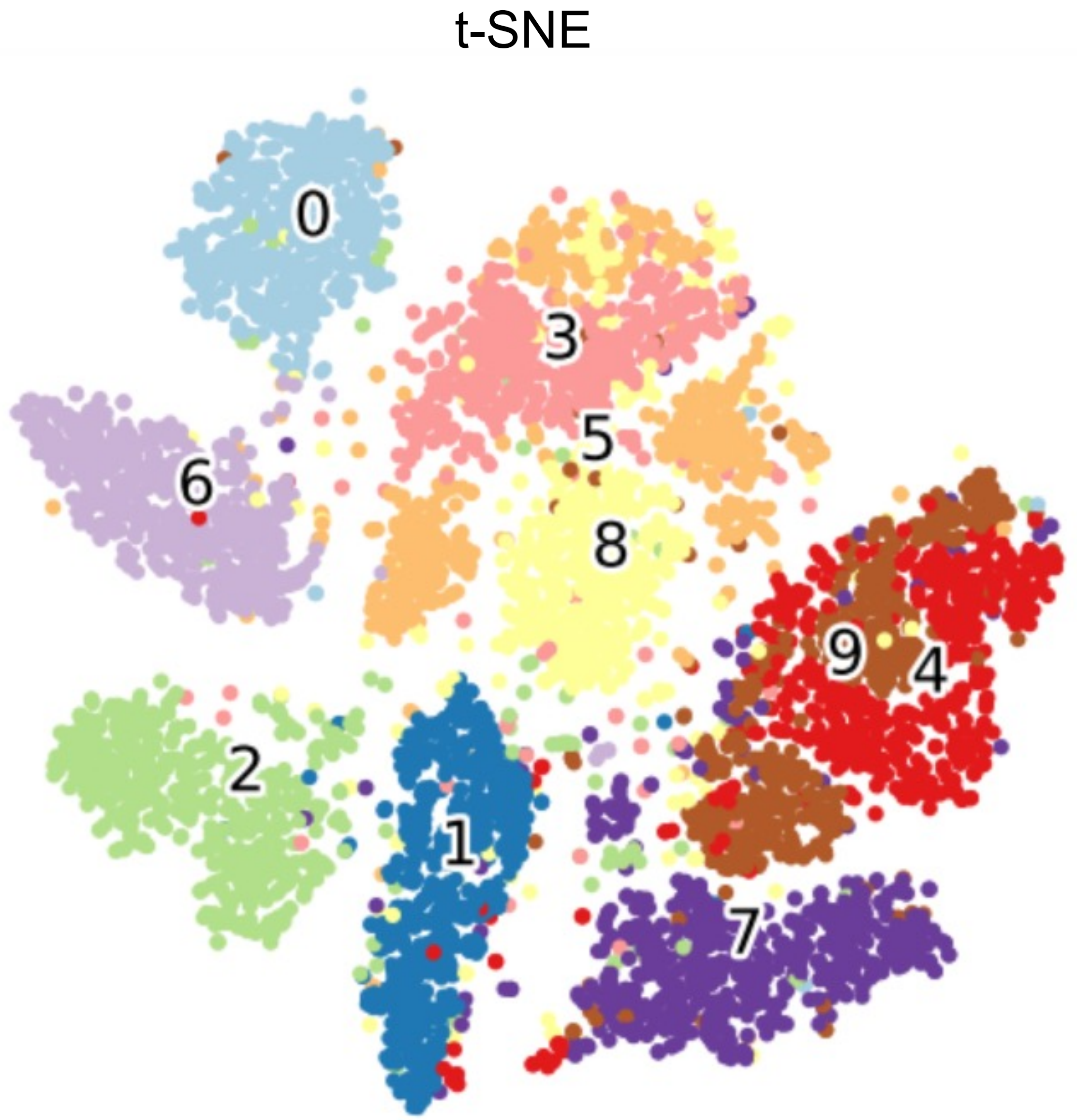
T-distributed Stochastic Neighbor Embedding (t-SNE)

- An unsupervised, randomized algorithm, used only for visualization

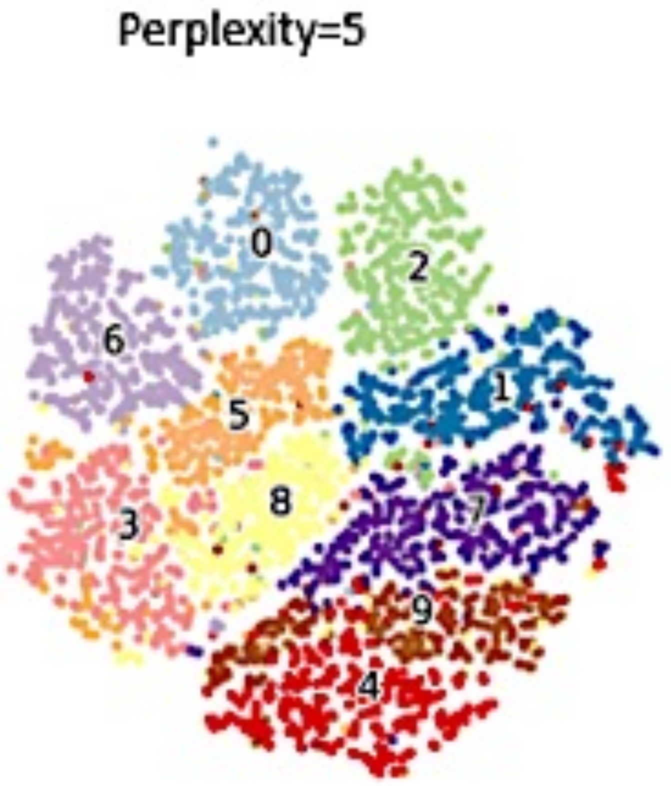
- Applies a non-linear dimensionality reduction technique where the focus is on keeping the very similar data points close together in lower-dimensional space.

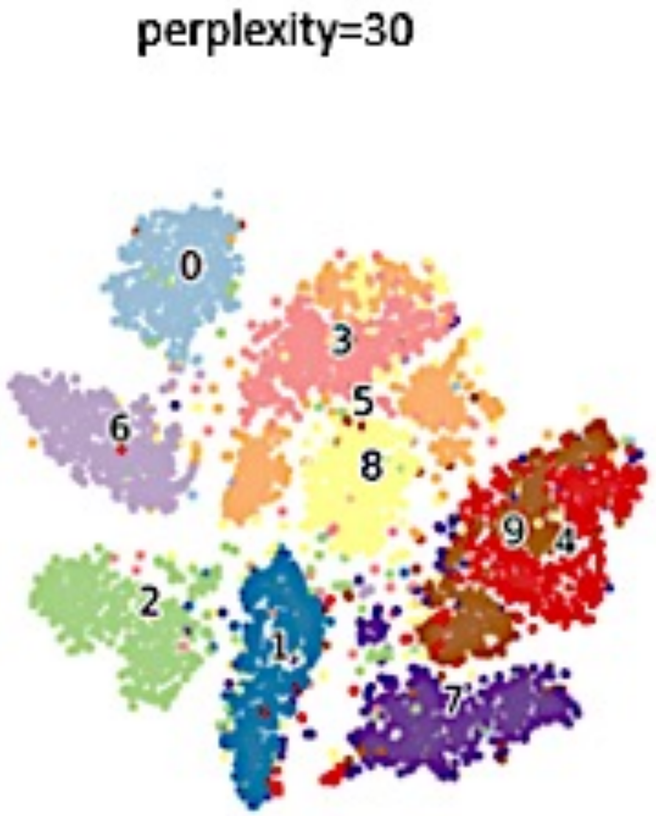
- Preserves the local structure of the data using student t-distribution to compute the similarity between two points in lower-dimensional space.

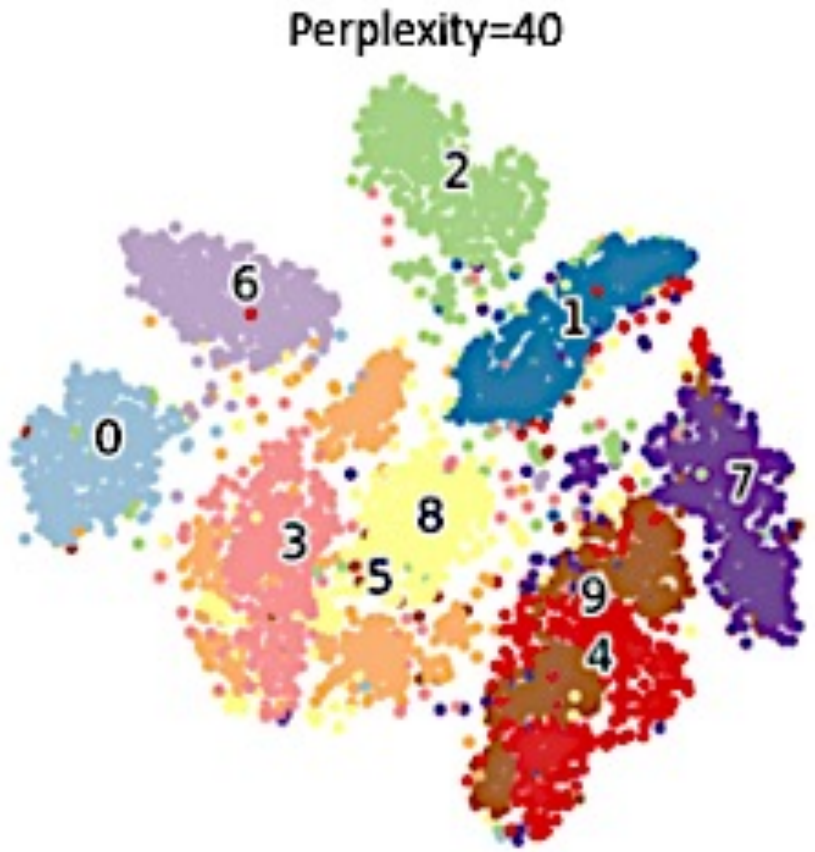




https://towardsdatascience.com/t-distributed-stochastic-neighbor-embedding-t-sne-bb60ff109561



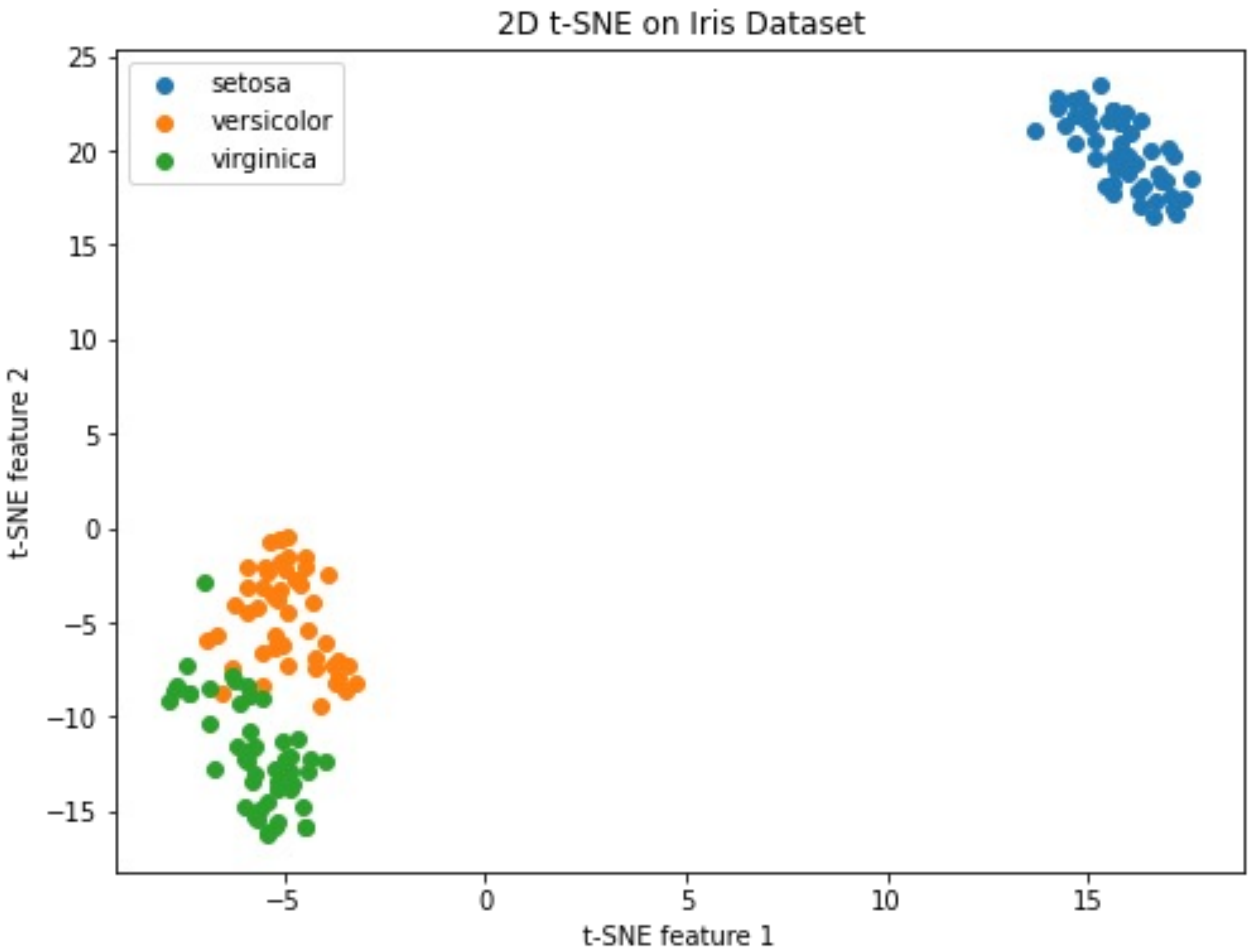




https://towardsdatascience.com/t-distributed-stochastic-neighbor-embedding-t-sne-bb60ff109561

```python
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.manifold import TSNE

# Load dataset
iris = datasets.load_iris()
X = iris.data
y = iris.target

# Initialize t-SNE
tsne = TSNE(n_components=2, random_state=0)

# Run t-SNE and get the transformed 2D representation
X_2d = tsne.fit_transform(X)
```

```python
# Create a scatter plot
plt.figure(figsize=(8, 6))

# Scatter plot for each class label
for i, label in enumerate(iris.target_names):
    plt.scatter(X_2d[y == i, 0], X_2d[y == i, 1], label=label)

plt.legend()
plt.xlabel('t-SNE feature 1')
plt.ylabel('t-SNE feature 2')
plt.title('2D t-SNE on Iris Dataset')

# Show the plot
plt.show()
```


2D t-SNE on Iris Dataset
setosa
versicolor
virginica
t-SNE feature 1
t-SNE feature 2
25
20
15
10
5
0
−5
−10
−15
−5
0
5
10
15

Uniform Manifold Approximation and Projection (UMAP)

UMAP has several advantages:

- **Speed**: UMAP is generally faster than other techniques like t-SNE, which makes it scalable to larger datasets.

- **Preservation of Global and Local Structure**: While t-SNE focuses on preserving local structure, UMAP maintains both the local and global aspects of data.

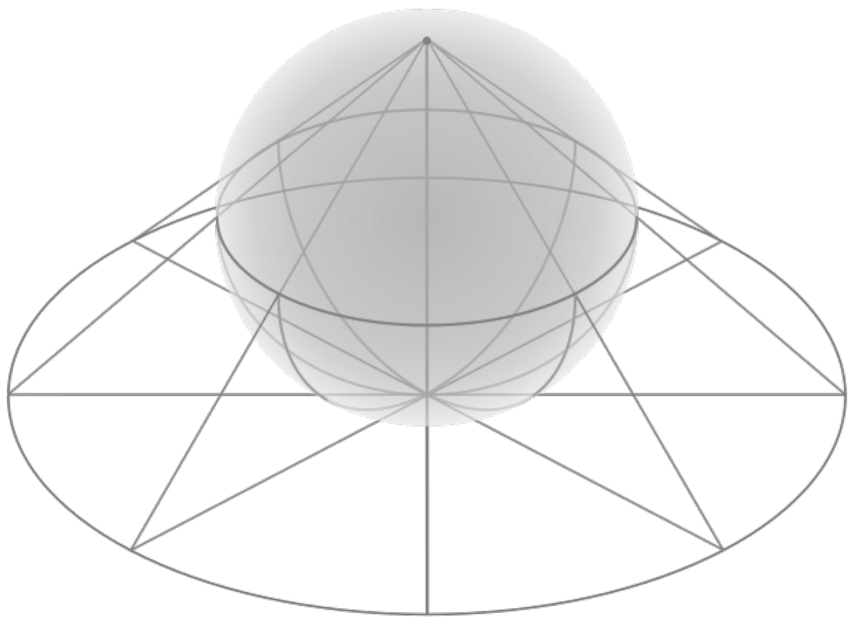
UMAP has several advantages:

- **Compatibility**: UMAP can be used in a wide range of applications, not just for visualization. It can be used for clustering, anomaly detection, and more.

- **Less Arbitrary Parameters**: UMAP has fewer parameters to tune, and they are often easier to interpret (like "minimum distance" and "number of neighbors").

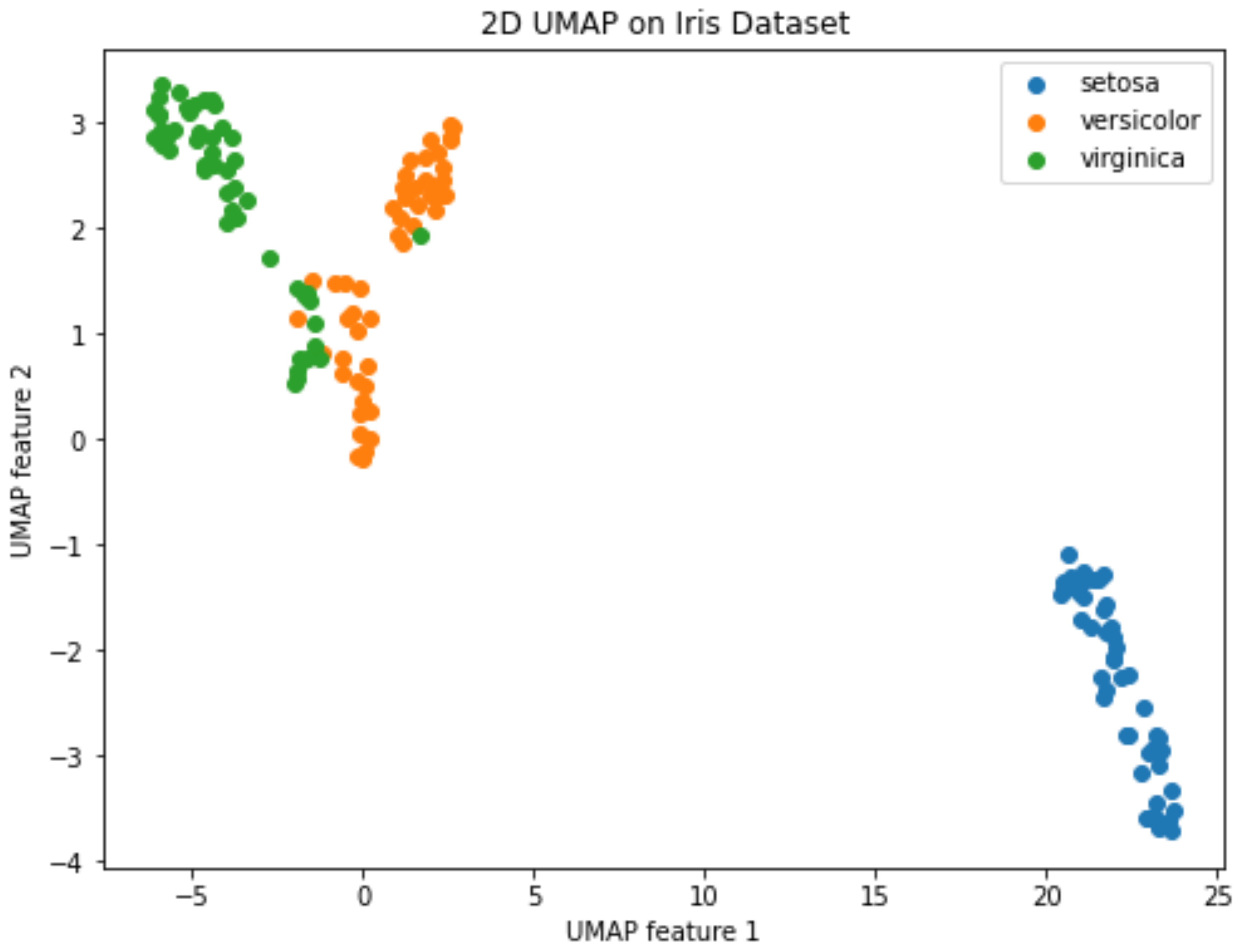
UMAP has several advantages:

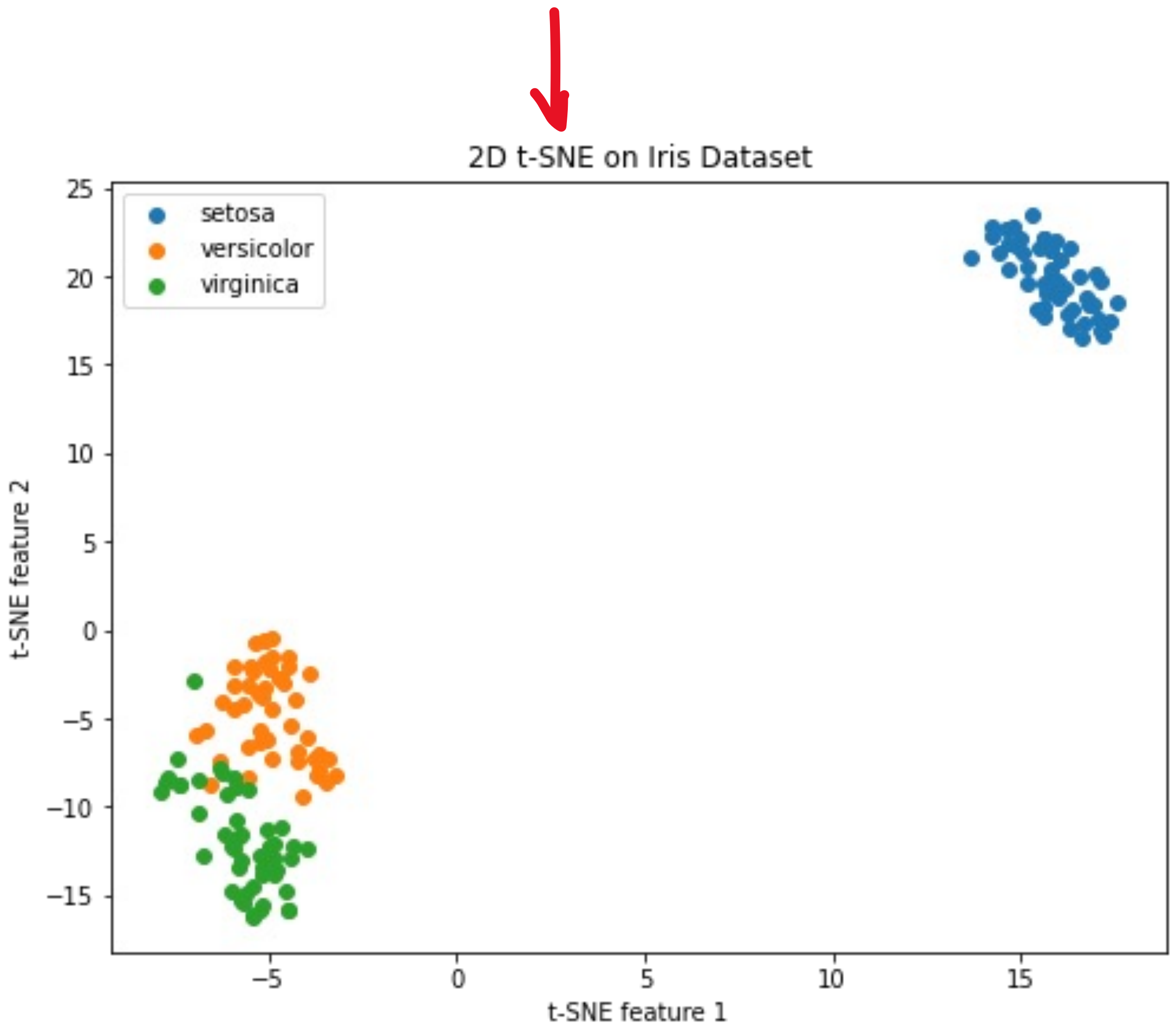
- **Theoretical Framework**: UMAP comes with a solid mathematical foundation, based on Riemannian geometry and algebraic topology.

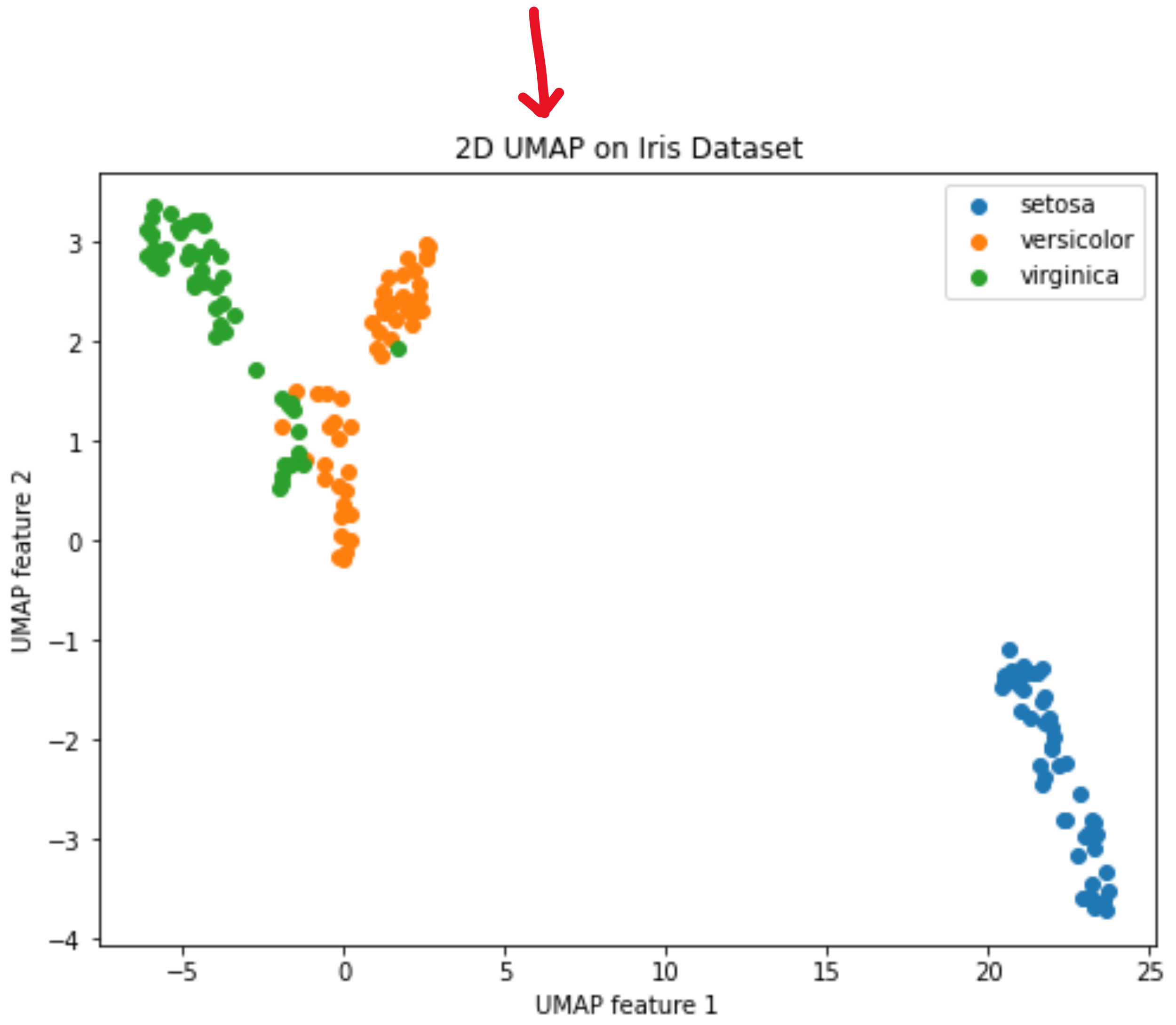


```python
import matplotlib.pyplot as plt
from sklearn import datasets
import umap

# Load dataset
iris = datasets.load_iris()
X = iris.data
y = iris.target

# Initialize UMAP
umap_model = umap.UMAP(n_neighbors=15, min_dist=0.1,
n_components=2)

# Run UMAP and get the transformed 2D representation
X_2d = umap_model.fit_transform(X)
```

```python
# Create a scatter plot
plt.figure(figsize=(8, 6))

# Scatter plot for each class label
for i, label in enumerate(iris.target_names):
    plt.scatter(X_2d[y == i, 0], X_2d[y == i, 1], label=label)

plt.legend()
plt.xlabel('UMAP feature 1')
plt.ylabel('UMAP feature 2')
plt.title('2D UMAP on Iris Dataset')

# Show the plot
plt.show()
```


2D UMAP on Iris Dataset
setosa
versicolor
virginica
UMAP feature 1
UMAP feature 2
−5
0
5
10
15
20
25
3
2
1
0
−1
−2
−3
−4


2D t-SNE on Iris Dataset
setosa
versicolor
virginica
t-SNE feature 1
t-SNE feature 2


2D UMAP on Iris Dataset
setosa
versicolor
virginica
UMAP feature 1
UMAP feature 2