EE 046202 - Technion - Unsupervised Learning & Data Analysis

[Tal Daniel (https://taldatech.github.io)](https://taldatech.github.io)

Tutorial 05 - Dimensionality Reduction - Principle Component Analysis (PCA), Kernels & KPCA

Agenda

- Motivation- Why Dimensionality Reduction?
- Dimensionality Reduction
- Principal Component Analysis (PCA) Recap)
  - PCA Algorithm
  - PCA for Compression
  - The Transpose Trick
  - Relation to SVD (Singular Value Decomposition)
- Kernels Motivation
- The Kernel Trick
- NLPCA- Nonlinear PCA)
  - Kernel PCA
- Recommended Videos
- Credits

In [1]:
```
# imports for the tutorial
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from sklearn.decomposition import PCA, KernelPCA
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_circles, make_moons
%matplotlib notebook
```

Motivation- Why Dimensionality Reduction?

- Discover Hidden Correlation/Topics - when we reduce dimensions, we sometimes discover **correlation between features**.
  - For example, we can notice two features that occur commonly together.
  - **Anomaly (Outlier) Detection**
- Remove Redundant and Noisy Features
  - Not all features are useful and sometimes harm the performance.
- Interpretation & Visualization
  - For example, when we reduce n-dimensional features to 2 or 3, we can plot them and **see the relationship with our eyes**.
- Easier Storage and Processing of the Data
  - Reduces **time and space complexity**.
  - Yields a more optimized process.
- Alleviates **The Curse of Dimensionality**
  - Fewer dimensions $\rightarrow$ less chance of *overfitting* $\rightarrow$ better generalization (recall what happens to classic ML models that become too complex).

Dimensionality Reduction

- Dimensionality reduction is the process of reducing the dimensionality of the feature space with consideration by obtaining a set of principal features.
  - Dimensionality reduction can be further broken into **feature selection** and **feature extraction**.
- Dimensionality Reduction vs. **Feature Selection**
  - Differs from feature selection in 2 ways:
    1. Instead of choosing subset of features, it **creates new features (dimensions) defined as functions over all features**.
    2. Does not consider **class labels**, just data points.
- Main Idea:
  - Given data points in d-dimensional space.
  - Project the data points into lower dimensional space while **preserving as much information as possible**.
    - For example, find the best 2D approximation to 3/4/104-D data.
  - In particular, choose the projection that minimizes the squared error in **reconstruction** of the original data.

Principal Component Analysis (PCA)

PCA is a method for reducing the dimensionality of data.

It uses simple matrix operations from linear algebra and statistics to calculate a projection of the original data into the same number or fewer dimensions.

PCA Goals:

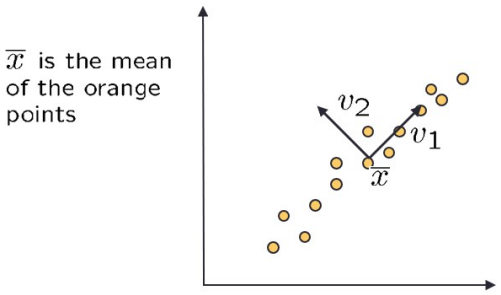
1. Find **linearly independent dimensions** (or basis of views) which can losslessly represent the data points.
2. Those newly found dimensions should allow us to predict/reconstruct the original dimensions. **The reconstruction/projection error should be minimized** (in Frobenius norm).

- More formally, PCA finds a new set of dimensions (or a set of basis of views) such that all the dimensions are **orthogonal** (and hence linearly independent) and ranked according to the variance of data along them.
- It means that the more important principal axis occurs first where **more important = more variance/more spread out data**.

Recap of some basics:

- **Variance** - a measure of the variability. Mathematically, it is the average squared deviation from the mean score. We use the following formula to compute variance: $var(x) = \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu_x)^2$ where μ_x is the mean.
- **Covaraince** - a measure of the extent to which corresponding elements from two sets of ordered data move in the same direction. We use the following formula to compute variance: $cov(x, y) = \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu_x)(y_i - \mu_y)$. Replace $\frac{1}{N}$ with $\frac{1}{N-1}$ for the *unbiased* estimation.
- **Covariance matrix** - includes the variance of dimensions on the main diagonal and the rest is the covariance between dimensions. If we have N data points (samples) with d dimensions for each sample and X is an dxN matrix, then: $Cov(X) = \frac{1}{N}(X - \mu_X)(X - \mu_X)^T$ (in PCA, we wish this matrix to be diagonal). We assume the data is centered, thus: $Cov(X) = \frac{1}{N}XX^T$. Replace $\frac{1}{N}$ with $\frac{1}{N-1}$ for the *unbiased* estimation.
- In the PCA case, multiplying by $\frac{1}{N-1}$ will not have much effect on the result, so in the following we will skip this step.

PCA Intuition



- Consider the variance along direction v (projection) among all the orange points:

$$var(v) = \sum_{orange\ points\ x} ||(x - \overline{x}) \cdot v||^2$$

- What is the unit vector v that **minimizes** the variance?
  - $\min_v(var(v)) = v_2$
- What is the unit vector v that **maximizes** the variance?
  - $\max_v(var(v)) = v_1$

- $var(v) = var((x - \overline{x})^T \cdot v) = \sum_x ||(x - \overline{x})^T \cdot v||^2 = \sum_x v^T (x - \overline{x})(x - \overline{x})^T v = v^T \left[\sum_x (x - \overline{x})(x - \overline{x})^T\right] v = v^T A v$
- Formally:

$$\max v^T A v$$
$$s.t\ ||v|| = 1,$$

  where $A = \sum_x (x - \overline{x})(x - \overline{x})^T = (X - \overline{\overline{X}})(X - \overline{\overline{X}})^T$

- **Solution**:
  - v_1 is eigenvector of A with the **largest** eigenvalue
  - v_2 is eigenvector of A with the **smallest** eigenvalue

PCA Algorithm

1. **Normalize/Standartize** (if we use features of different scales, we may get misleading components) and center the data. Given data $X \in \mathcal{R}^{m \times N}$, where m is the number of features and N is the number of samples, normalization:

   $$\tilde{X} = X - \overline{X}$$

   Standartization:

   $$\tilde{X} = \frac{X - \overline{X}}{\bar{\sigma}_x}$$

   Where $\bar{\sigma}_x$ is the empirical standard deviation (the square root of the empirical variance).
2. Calculate the empirical covariance matrix X of data points:

   $$P = \tilde{X}\tilde{X}^T \in \mathcal{R}^{m \times m}$$

   - Note that it is usually better to normalize:

   $$P = \frac{1}{N-1}\tilde{X}\tilde{X}^T$$
3. Calculate eigenvectors and corresponding eigenvalues.
4. Sort the eigenvectors according to their eigenvalues in decreasing order.
5. Choose first k largest eigenvectors and that will be the new k dimensions.
6. Transform the original d dimensional data points into k dimensions.

Example - PCA on Breast Cancer Dataset

The Breast Cancer Wisconsin (Diagnostic) Data Set

This dataset contains features of breast cancer and classify them to benign/malignant. Features are computed from a digitized image of a fine needle aspirate (FNA) of a breast mass. They describe characteristics of the cell nuclei present in the image.

- We will take the first 3 features, and reduce the dimensionality to 2 using PCA.

In [2]:
```
# load the data
dataset = pd.read_csv('./datasets/cancer_dataset.csv')
# print the number of rows in the data set
number_of_rows = len(dataset)
print('Number of rows in the dataset: {}'.format(number_of_rows))
## Show a sample 10 rows
dataset.sample(10)
```

```
Number of rows in the dataset: 569
```

Out[2]:

	id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concav
168	8712766	M	17.47	24.68	116.10	984.6	0.10490	0.16030	
507	91544002	B	11.06	17.12	71.25	366.5	0.11940	0.10710	
354	901011	B	11.14	14.07	71.24	384.6	0.07274	0.06064	
477	911673	B	13.90	16.62	88.97	599.4	0.06828	0.05319	
356	9010259	B	13.05	18.59	85.09	512.0	0.10820	0.13040	
253	8860702	M	17.30	17.08	113.00	928.2	0.10080	0.10410	
41	855563	M	10.95	21.35	71.90	371.1	0.12270	0.12180	
260	887549	M	20.31	27.06	132.90	1288.0	0.10000	0.10880	
83	8611792	M	19.10	26.29	129.10	1132.0	0.12150	0.17910	
488	913512	B	11.68	16.17	75.49	420.5	0.11280	0.09263	

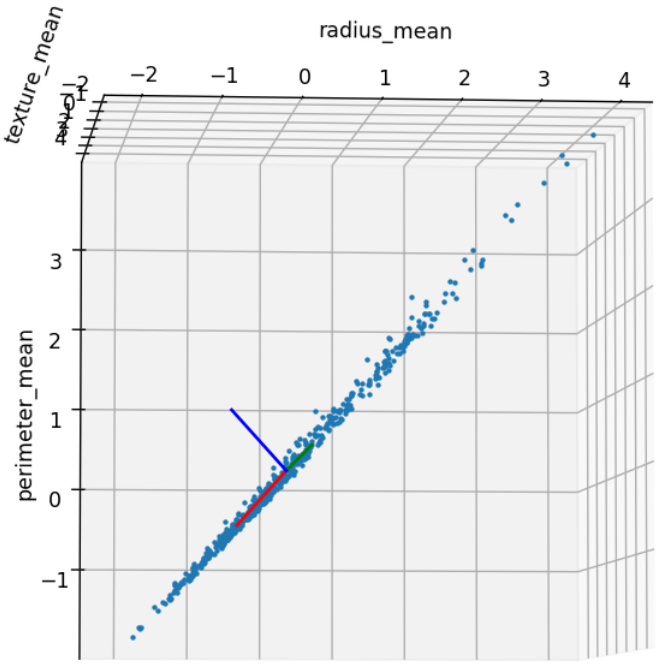
10 rows × 33 columns

In [9]:
```
# take only the first 3 features
x = dataset[['radius_mean', 'texture_mean', 'perimeter_mean']].values
# standartize the data (centering and normalizing), features of different scale!
# note: you can also use scikit-learn's StandardScaler()
x -= x.mean(axis=0, keepdims=True)
x /= x.std(axis=0, keepdims=True)
# calculate the covariance matrix
A = x.T @ x  # x in [N x m]
# calculate eigenvalues and eigenvectors
# NOT ordered in decreasing order
d, v = np.linalg.eig(A)
# sort by decreasing order
v = v[:,np.argsort(-d)]
d = d[np.argsort(-d)]
print("eigenvalues:")
print(d.astype(np.float16))
# the reconstruction of x would be x ~ X @ V @ V.T
# take the 2 most dominant directions
print("projection - dimension reduction (3 to 2):")
x_proj = x @ v[:, :-1]
print(x_proj)
```

```
eigenvalues:
[1.24e+03 4.66e+02 1.21e+00]
projection - dimension reduction (3 to 2):
[[-0.80196001  2.54048135]
 [-2.18555934  1.23675759]
 [-2.23789966  0.38704729]
 ...
 [-1.65154304 -1.54971556]
 [-3.36804781 -1.19009381]
 [ 1.93933426 -2.07217819]]
```

In [10]:
```
def plot_pca():
    # plot
    fig = plt.figure(figsize=(8, 8))
    ax = fig.add_subplot(1, 1, 1, projection='3d')
#     ax.axis('equal')
    ax.set_xlabel('radius_mean',)
    ax.set_ylabel('texture_mean')
    ax.set_zlabel('perimeter_mean')
    ax.plot(x[:, 0], x[:, 1], x[:, 2], '.', markersize=3)
    ax.plot([0, v[0, 0]], [0, v[1, 0]], [0, v[2, 0]], 'r') # most dominant eigenvector
    ax.plot([0, v[0, 1]], [0, v[1, 1]], [0, v[2, 1]], 'g')
    ax.plot([0, v[0, 2]], [0, v[1, 2]], [0, v[2, 2]], 'b')
```

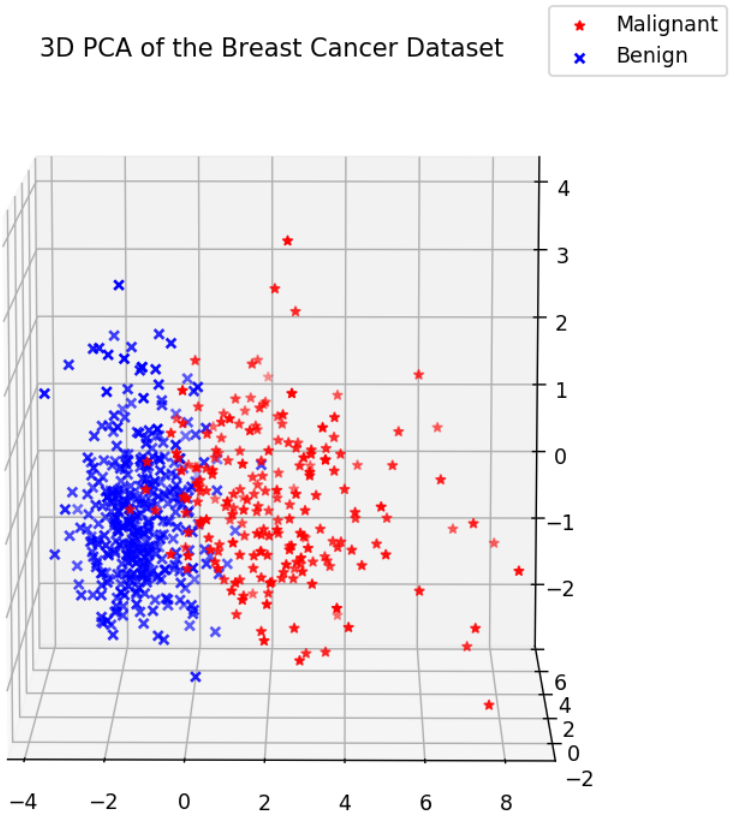
In [11]:
```
%matplotlib notebook
plot_pca()
```



In [12]:
```
# using scikit-learn
scaler = StandardScaler()
X = dataset[['radius_mean', 'texture_mean',
             'perimeter_mean', 'area_mean',
             'smoothness_mean', 'compactness_mean', 'concavity_mean']].values
X = scaler.fit_transform(X)
y = dataset['diagnosis'].values == 'M'
pca = PCA(n_components=3)
X_3d = pca.fit_transform(X)
```

In [13]:
```
def plot_sk_pca():
    # plot
    fig = plt.figure(figsize=(8, 8))
    ax = fig.add_subplot(1, 1, 1, projection='3d')
    ax.scatter(X_3d[y,0], X_3d[y, 1], X_3d[y, 2], color='r', marker='*', label='Malignant')
    ax.scatter(X_3d[~y,0], X_3d[~y, 1], X_3d[~y, 2], color='b', marker='x', label='Benign')
    ax.grid()
    ax.legend()
    ax.set_title("3D PCA of the Breast Cancer Dataset")
```

In [14]:
```
plot_sk_pca()
```



PCA for Compression

- The **projection matrix** is a matrix composed of the data projected onto the top-K eigenvectors.
- To get a better understanding of the dimensionality reduction quality, we observe the trade-off between the **compression and the reconstruction error**.
  - The more compression (that is, lower dimension) the larger the reconstruction error and the representation quality is degraded (as our new features don't represent the original data faithfully).

- Measuring the normalized reconstruction error:
  - Denote the top-K eigenvector matrix: $W_k \in \mathcal{R}^{m \times k}$
  - The projection: $Z = XW_K \in \mathcal{R}^{n \times k}$
  - The reconstruction: $\tilde{X} = ZW_k^T = XW_kW_k^T \in \mathcal{R}^{N \times m}$
  - Measure the error by the **Matrix Norm: Frobenius Norm**:

$$||M||_F^2 = \sum_{ij} M_{ij}^2 \rightarrow ||A - B||_F^2 = \sum_{ij} (A_{ij} - B_{ij})^2$$

  - The normalized reconstruction error:

$$err_k = \frac{||XW_kW_k^T - X||_F^2}{||X||_F^2}$$

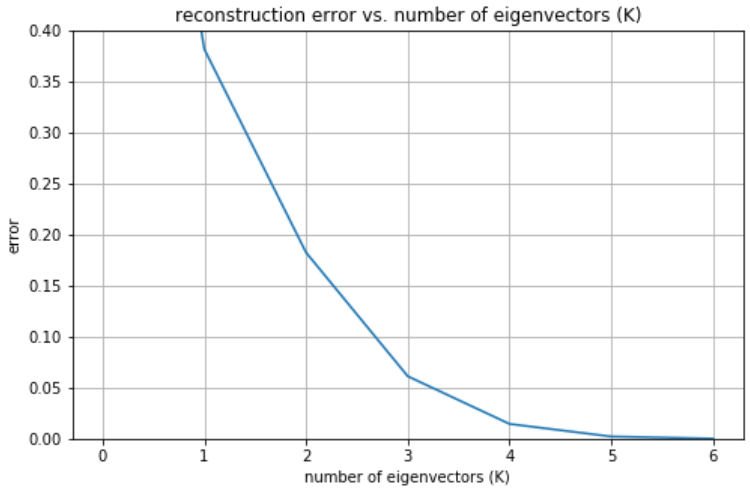
- How to pick k?
  - As a rule of thumb we take the amount of eigenvectors that allows no more than 1% reconstruction error.

In [9]:
```
X_normalized = X - X.mean(axis=0, keepdims=True)
X_norm = np.linalg.norm(X_normalized, ord='fro')  # ord='fro' for Frobenius norm
# calculate the covariance matrix
A = X_normalized.T @ X_normalized  # x in [N x m]
d, v = np.linalg.eig(A)
# sort by decreasing order
v = v[:,np.argsort(-d)]
d = d[np.argsort(-d)]
for k in range(1, X_normalized.shape[1] + 1):
    Z = X_normalized @ v[:,:k]
    err = np.square(np.linalg.norm(Z @ v[:,:k].T - X_normalized, ord='fro') / X_norm)
    print("number of eigenvectors (k): {}, reconstruction error: {}".format(k, err))
```

```
number of eigenvectors (k): 1, reconstruction error: 0.3817772746473096
number of eigenvectors (k): 2, reconstruction error: 0.18251649332420763
number of eigenvectors (k): 3, reconstruction error: 0.061153584222750024
number of eigenvectors (k): 4, reconstruction error: 0.014432620496369605
number of eigenvectors (k): 5, reconstruction error: 0.002089071617164215
number of eigenvectors (k): 6, reconstruction error: 4.1709202569973447e-05
number of eigenvectors (k): 7, reconstruction error: 3.4934265607613812e-31
```

In [12]:
```
def plot_pca_recon_error(X, v, d):
    k_s = list(range(X.shape[1]))
    X_norm = np.linalg.norm(X, ord='fro')
    errs = []
    for k in k_s:
        Z = X @ v[:,:k]
        err = np.square(np.linalg.norm(Z @ v[:,:k].T - X, ord='fro') / X_norm)
        errs.append(err)
    fig = plt.figure(figsize=(8, 5))
    ax = fig.add_subplot(111)
    ax.plot(k_s, errs)
    ax.grid()
    ax.set_xlabel("number of eigenvectors (K)")
    ax.set_ylabel("error")
    ax.set_title("reconstruction error vs. number of eigenvectors (K)")
    ax.set_ylim([0, 0.4])
```

In [13]:
```
%matplotlib inline
plot_pca_recon_error(X_normalized, v, d)
```



The Transpose Trick

- What happens when the number of features is very large and much larger than the number of samples, that is, $m >> N$?
  - Calculating the $m \times m$ covariance matrix is computationally expensive ($O(m^2N)$).
- **The Transpose Trick**: ($X \in \mathcal{R}^{m \times N}$)
  - Instead of calculating the eigenvalues and eigenvectors of $\frac{1}{N}XX^T$ we compute the eigenvalues and eigenvectors of
    $$\frac{1}{m}X^TX$$
  - *Why???*
    - If v is an eigenvector of XX^T, then:
      $$XX^Tv = \lambda v$$
    - Left-multiplying by X^T, we get
      $$X^TX(X^Tv) = \lambda(X^Tv)$$
      - $\rightarrow X^Tv$ is an **eigenvector** of X^TX with **eigenvalue** λ.
    - In order to compute v, which is really what we want:
      - Denote the eigenvector of X^TX by w.
      - We get:
        $$Xw = XX^Tv = \lambda v \rightarrow v = \lambda^{-1}Xw$$

The Relationship Between PCA & SVD

- The PCA viewpoint requires that one compute the eigenvalues and eigenvectors of the covariance matrix, which is the product XX^T, where X is the data matrix. Since the covariance matrix is symmetric, the matrix is diagonalizable, and the eigenvectors can be normalized such that they are orthonormal: $XX^T = WAW^T$
- On the other hand, applying SVD to the data matrix X as follows: $X = U\Sigma V^T$, and attempting to construct the covariance matrix from this decomposition gives:
  $$XX^T = (U\Sigma V^T)(U\Sigma V^T)^T = U\Sigma^2U^T$$
  the last transition is due to V being orthonormal ($VV^T = I$). Thus, the square roots of the eigenvalues of XX^T are the singular values of X.
- Using the SVD to perform PCA makes much better sense numerically than forming the covariance matrix to begin with, since the formation of $XX^\top$ can cause loss of precision. But performing SVD is slower.

PCA as Dimensionality Reduction Technique

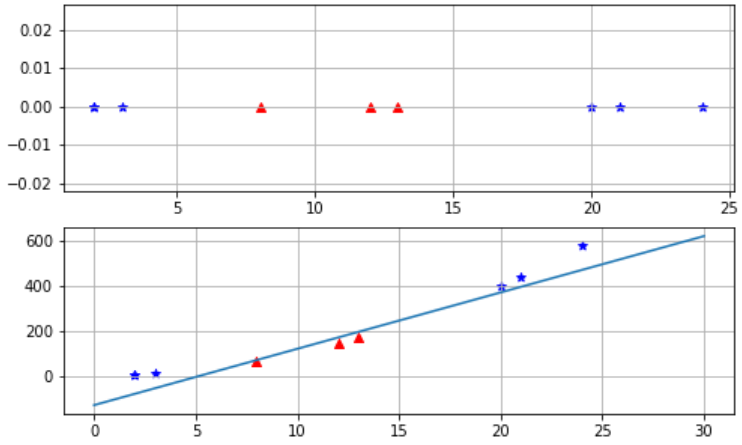
- **Pro**: Optimal reconstruction error in Frobenius norm.
- **Con**: Interpretability problem - **features lose their previous meaning**.
  - A singular vector specifies a linear combination of all input columns or rows
  - PCA is **sensitive to outliers** since it is minimizing l_2 norms. The squaring of deviations from the outliers, they will **dominate the total norm** and therefore will drive the PCA components.
- **When will PCA work?**
  - PCA assumes **linear** realtionships among variables.
  - Clouds of points in p-dimensional space has linear dimensions that can be effectively summarized by the principal axes.
  - If the structure in the data is **non-linear** (the cloud of points twists and curves its way through p-dimensional space), the principal axes will not be an efficient and informative summary of the data.

Kernels Motivation

- The main shortcoming of PCA is that it is unable to capture nonlinear structures in the data.
- Consider the following example of linearly inseparatable 1-D set of exmaples and then extracting polynimial (second order) features (**feature extraction**):

In [14]:
```
def plot_kernel_example():
    x_1 = np.random.randint(0,6, size=(3,))
    x_2 = np.random.randint(8,14, size=(3,))
    x_3 = np.random.randint(20,25, size=(3,))

    x_1_p = x_1 ** 2
    x_2_p = x_2 ** 2
    x_3_p = x_3 ** 2

    x_class = np.linspace(0, 30, 400)
    y_class = 25 * x_class - 130

    fig = plt.figure(figsize=(8,5))
    ax_1 = fig.add_subplot(211)
    ax_1.scatter(x_1, np.zeros_like(x_1), marker='*', color='b')
    ax_1.scatter(x_2, np.zeros_like(x_2), marker='^', color='r')
    ax_1.scatter(x_3, np.zeros_like(x_3), marker='*', color='b')
    ax_1.grid()
    ax_2 = fig.add_subplot(212)
    ax_2.scatter(x_1, x_1_p, marker='*', color='b')
    ax_2.scatter(x_2, x_2_p, marker='^', color='r')
    ax_2.scatter(x_3, x_3_p, marker='*', color='b')
    ax_2.plot(x_class, y_class)
    ax_2.grid()
```

In [15]:
```
plot_kernel_example()
```



- Adding polynomial features is simple to implement and can work great with all sorts of ML algorithms.
- At a **low polynomial** degree it cannot deal with more complex datasets.
- At a **high polynomial** degree there are a lot of features, which makes the computation very slow.
  - Computation in the feature space can be costly because it is high dimensional (even go to infinity).
- **The Kernel Trick** comes to the rescue!
  - It makes it possible to get the same result as if you added many features (even in high dimension), **without actually adding them**!
    - So there is no computational disaster resulting from the large number of features.

The Kernel Trick

- When the **data points**, $\{x^{(i)}\}_{i=1}^{m}$, only appear as **inner (dot) products** - $(x^{(i)})^T x^{(j)}$, we can apply the kernel trick.
- As long as we can calculate the inner product in the **feature space**, we don't need the mapping explicitly.
  - The kernel function, denoted $K(x^{(i)}, x^{(j)}) = \phi(x^{(i)})^T \phi(x^{(j)})$, stands for the inner product in the feature space, and can easily plugged into the optimization.
  - If we could find a function $K(x^{(i)}, x^{(j)})$ that is simple enough, we can actually save us the huge number of calculations required to calculate $\phi(x^{(i)})^T \phi(x^{(j)})$, or even prevent us from the feature extraction itself, $\phi(x)$, this is the **kernel trick**.

- There are no restrictions on the form of $x^{(i)}, x^{(j)}$, but there are restrictions on the kernel function $K(x^{(i)}, x^{(j)})$.
  - $K(x^{(i)}, x^{(j)})$ must satisfy the **Mercer Condition** - the $n \times n$ kernel matrix $K(x^{(i)}, x^{(j)})$ is **PSD**.
  - This implies that the quadratic optimization is convex.
    - It has a uniuqe solution and can be solved efficiently
- **Mercer's Theorem**: *Suppose that $k(x, y)$ is a symmetric real valued function such that $k(x, y) \leq C$ for some $C > 0$. Suppose the linear operator*

$$\mathcal{L}(f)(x) = \int_{\mathcal{R}^D} k(x, y) f(y) dy$$

  is PSD, that is

$$\int \int_{\mathcal{R}^D} k(x, y) f(y) dy dx \geq 0$$

  Let ψ_i be the normalized orthogonal eigenfunctions of $\mathcal{L}$, associated with eigenvalues $\lambda_i > 0$, sorted in non-decreasing order, and let m be the number of non-zero eigenvalues.
- Then, the sequence of eigenvalues is summable, namely $\sum_i \mid \lambda_i \mid < \infty$, and the kernel can be expressed as

$$k(x, y) = \sum_{i=1}^{m} \lambda_i \psi_i(x) \psi_i(y)$$

- *We thus see that by choosing $\phi_i(x) = \sqrt{\lambda_i} \psi_i(x)$, a feature based representation is obtained.*

 The Kernel Trick - Polynomial Kernel Example

- We will now demonstrate the kernel trick for a 2^{nd}-degree polynomial mapping.
- Suppose the feature mapping (or extraction) is defined as follows:

$$\phi(u) = \phi(\begin{bmatrix} u_1 \\ u_2 \end{bmatrix}) = \begin{bmatrix} 1 \\ \sqrt{2}u_1 \\ \sqrt{2}u_2 \\ \sqrt{2}u_1 u_2 \\ u_1^2 \\ u_2^2 \end{bmatrix}$$

- The inner product $\phi(u)^T \phi(v)$:

$$\phi(u)^T \phi(v) = 1 + 2u_1 v_1 + 2u_2 v_2 + 2u_1 u_2 v_1 v_2 + u_1^2 v_1^2 + u_2^2 v_2^2 = (1 + u_1 v_1 + u_2 v_2)^2 = (1 + u^T v)^2$$

  - **HOW ABOUT THAT?** - We just found out that we can calculate $\phi(u)^T \phi(v)$ just by calculating the input inner product $u^T v$ and plugging it in $(1 + u^T v)^2$, thus saving us the whole feature mapping calculation process!
- We define **the 2^{nd} polynomial kernel**:

$$K(u, v) = (1 + u^T v)^2$$

  - No need to calculate $\phi(u)^T \phi(v)$ explicitly!
  - The use of kernel function to avoid computing $\phi(\cdot)$ explicitly is known as **the kernel trick**.

 Other Kernel Functions

- Linear kernel:

$$K(u, v) = u^T v$$

- Polynomial kernel with degree d:

$$K(u, v) = (\alpha u^T v + \beta)^d$$

  - The feature space is all monomials up to degree d

- **Radial Basis Function** (Gaussian RBF) kernel with width σ:

$$K(u,v) = e^{-\frac{||u-v||^2}{2\sigma^2}} = e^{-\gamma||u-v||^2} = exp(-\gamma||u-v||^2)$$

  - It is also a type of a *similarity function* that measures *how far are two points?*
  - RBF covers the space with "balls" of a fixed radius, where the centers are the support vectors (in SVM).
  - The radius of the "balls" is determined by the parameter $\gamma = \frac{1}{2\sigma^2}$
    - A **smaller** γ means a larger radius, a lower "model complexity"
    - A **larger** γ means a smaller radius, a finer grain coverage which may possibly lead to overfitting.
  - The feature space is **infinite-dimensional** (expand the exponential to an infinite series using Taylor series).
- Sigmoid kernel, with parameters κ and θ:

$$K(u,v) = tanh(\kappa u^T v + \theta)$$

  - It does not satisfy the *Mercer* condition on all κ, θ.

- Techniques for constructing new kernels (from *Pattern Recognition and Machine Learning*, C.Bishop):

Given valid kernels $k_1(\mathbf{x}, \mathbf{x}')$ and $k_2(\mathbf{x}, \mathbf{x}')$, the following new kernels will also be valid:

$$
\begin{aligned}
k(\mathbf{x}, \mathbf{x}') &= c k_1(\mathbf{x}, \mathbf{x}') && (6.13)\\
k(\mathbf{x}, \mathbf{x}') &= f(\mathbf{x}) k_1(\mathbf{x}, \mathbf{x}') f(\mathbf{x}') && (6.14)\\
k(\mathbf{x}, \mathbf{x}') &= q\left(k_1(\mathbf{x}, \mathbf{x}')\right) && (6.15)\\
k(\mathbf{x}, \mathbf{x}') &= \exp\left(k_1(\mathbf{x}, \mathbf{x}')\right) && (6.16)\\
k(\mathbf{x}, \mathbf{x}') &= k_1(\mathbf{x}, \mathbf{x}') + k_2(\mathbf{x}, \mathbf{x}') && (6.17)\\
k(\mathbf{x}, \mathbf{x}') &= k_1(\mathbf{x}, \mathbf{x}') k_2(\mathbf{x}, \mathbf{x}') && (6.18)\\
k(\mathbf{x}, \mathbf{x}') &= k_3\left(\phi(\mathbf{x}), \phi(\mathbf{x}')\right) && (6.19)\\
k(\mathbf{x}, \mathbf{x}') &= \mathbf{x}^{\mathrm{T}} \mathbf{A} \mathbf{x}' && (6.20)\\
k(\mathbf{x}, \mathbf{x}') &= k_a(\mathbf{x}_a, \mathbf{x}_a') + k_b(\mathbf{x}_b, \mathbf{x}_b') && (6.21)\\
k(\mathbf{x}, \mathbf{x}') &= k_a(\mathbf{x}_a, \mathbf{x}_a') k_b(\mathbf{x}_b, \mathbf{x}_b') && (6.22)
\end{aligned}
$$

where $c > 0$ is a constant, $f(\cdot)$ is any function, $q(\cdot)$ is a polynomial with nonnegative coefficients, $\phi(\mathbf{x})$ is a function from $\mathbf{x}$ to $\mathbb{R}^M$, $k_3(\cdot, \cdot)$ is a valid kernel in $\mathbb{R}^M$, $\mathbf{A}$ is a symmetric positive semidefinite matrix, $\mathbf{x}_a$ and $\mathbf{x}_b$ are variables (not necessarily disjoint) with $\mathbf{x} = (\mathbf{x}_a, \mathbf{x}_b)$, and k_a and k_b are valid kernel functions over their respective spaces.

Exercise - Valid Kernels

Let $k_1(x,y)$ and $k_2(x,y)$ be valid kernels according to Mercer's theorem (that is, they represent an inner product in some transformed space). Prove that the following functions are valid kernels as well:

1. Prove:

$$k(x,y) = k_1(x,y) + k_2(x,y)$$

2. Prove:

$$k(x,y) = f(x)k_1(x,y)f(y) \text{ where } f : \mathcal{R}^D \to \mathcal{R}$$

Solution

The basic trick here is using the initial knowledge that both kernels are valid according to Mercer's theorem.

Section 1

- Prove for:

$$k(x,y) = k_1(x,y) + k_2(x,y)$$

Using Mercer's theorem we know that there exists mappings $\phi_1(x)$ and $\phi_2(x)$ such that:

$$k_1(x,y) = \phi_1(x)^T \phi_1(y)$$
$$k_2(x,y) = \phi_2(x)^T \phi_2(y)$$

Thus,

$$\begin{aligned} k(x,y) &= k_1(x,y) + k_2(x,y) \\ &= \phi_1(x)^T \phi_1(y) + \phi_2(x)^T \phi_2(y) \\ &= [\phi_1(x), \phi_2(x)]^T [\phi_1(y), \phi_2(y)] = \Phi(x)^T \Phi(y) \end{aligned}$$

- $[\phi_1(x), \phi_2(x)]$ is a concatentanion of the vectors $\phi_1(x), \phi_2(x)$:

$$[\, \phi_{11}(x) \quad \dots \quad \phi_{1n}(x) \quad \phi_{21}(x) \quad \dots \quad \phi_{2m}(x) \,]^T$$

- Thus, $k(x,y)$ represents an inner product in some transformed space.

Section 2

- Prove for:

$$k(x,y) = f(x) k_1(x,y) f(y) \;\; \text{where} \;\; f : \mathcal{R}^D \to \mathcal{R}$$

Again, using Mercer's theorem:

$$\begin{aligned} k(x,y) &= f(x) k_1(x,y) f(y) \\ &= f(x) \phi_1(x)^T \phi_1(y) f(y) \\ &= \Phi(x)^T \Phi(y) \end{aligned}$$

where $\Phi(x) = f(x)\phi_1(x)$.

- Recall that $\phi_1(x)$ is a vector and $f(x)$ is a **scalar**.
- Thus, $k(x,y)$ represents an inner product in some transformed space.

Nonlinear PCA - NLPCA (& Kernel PCA)

- The main shortcoming of PCA is that it is **unable to capture nonlinear structures in the data**.
- The basic idea behind Kernel PCA is that by embedding the data into a high-dimensional space for which linear sub-spaces capture the relavant structures.
- Mathematically:

$$x \in \mathcal{R}^D \xrightarrow{\text{nonlinear}} \phi(x) \in \mathcal{R}^m \xrightarrow{\text{linear}} y \in \mathcal{R}^d, m \geq D > d$$

(and usually $m >> D$)

- The basic steps of **Nonlinear PCA**:
  - Using the mapping $\phi(x)$, we map each data point $x_i \to \phi(x_i),\ i = 1, \ldots, n$.
  - As in regular PCA, we center (or normalize/standartize) the data. We define:
$$\Phi = \left(\phi(x_1) - \bar{\phi}, \ldots, \phi(x_n) - \bar{\phi}\right) \in \mathcal{R}^{m \times n}$$
$$\bar{\phi} = \frac{1}{n}\sum_{j=1}^{n}\phi(x_j)$$
  - Define the sample covariance matrix:
$$\hat{P}_{\phi(x)} = \frac{1}{n}\sum_{j=1}^{n}\left(\phi(x_j) - \bar{\phi}\right)\left(\phi(x_j) - \bar{\phi}\right)^T = \frac{1}{n}\Phi\Phi^T \in \mathcal{R}^{m \times m}$$
  - Perform the *eigen-decomposition* of the sample covariance matrix:
$$\hat{P}_{\phi(x)}u_i = \lambda_i u_i,\ i = 1, \ldots, m$$
    - The d *nonlinear* Principal Components (PCs):
$$y_i = u_i^T\left(\phi(x) - \bar{\phi}\right) \in \mathcal{R},\ u_i \in \mathcal{R}^m,\ i = 1, \ldots, d$$

- **Compuational cost**:
  - Computing the eigenvectors of a $m \times m$ matrix requires (in general) $O(m^3)$ steps - very expensive for large m!
- **Note** - the features $\phi(x_i)$ need to be centered even if the x_i are centered, since the mapping ϕ is **nonlinear**.

- Reducing the computational cost using the **Transpose Trick**:
  - The dimension m of the feature space may be very large which may be very expensive computationally.
  - Using the *transpose trick* which is based on the relation between the eigen-decomposition of $\Phi\Phi^T$ and $\Phi^T\Phi$, we can reduce the cost.
    - Recall the the **eigenvalues** of $\Phi\Phi^T$ and $\Phi^T\Phi$ are equal.
  - Doing the same derivation as before (look up...) we get that if w is an eigen-vector of $\Phi^T\Phi$ and v is an eigen-vector of $\Phi\Phi^T$ then:
$$u = \lambda^{-1}\Phi w$$
    - It is much cheaper to compute as the number of samples $n << m$.
  - The d *nonlinear* Principal Components (PCs):
$$y_i = u_i^T\left(\phi(x) - \bar{\phi}\right) = \lambda_i^{-1}w_i^T\Phi^T\left(\phi(x) - \bar{\phi}\right) \in \mathcal{R},\ u_i \in \mathcal{R}^m,\ i = 1, \ldots, d$$
$$Y = \Lambda^{-1}W^T\Phi^T\Phi \in \mathcal{R}^{d \times n}$$

Kernel PCA

- Notice that the nonlinear PCs only require the computation of *inner products* between the features ϕ. This indicates that we can use **the kernel trick** to calculate these inner products.
- We define the kernel and *centered kernel functions*:
$$k(x, y) = \phi(x)^T\phi(y)$$
$$\tilde{k}(x, y) = \left(\phi(x) - \bar{\phi}\right)^T\left(\phi(y) - \bar{\phi}\right)$$
  - Note that:
$$\tilde{k}(x, y) = k(x, y) - \frac{1}{n}\sum_{j=1}^{n}k(x, x_j) - \frac{1}{n}\sum_{i=1}^{n}k(x_i, y) + \frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n}k(x_i, y_j)$$
  - Define the kernel and centered *kernel matrices*, which are defined w.r.t. the data points:
$$K = \left[k_{ij}\right],\ \ k_{ij} = k(x_i, x_j)$$
$$\tilde{K} = \Phi^T\Phi = \left[\tilde{k}_{ij}\right],\ \ \tilde{k}_{ij} = \tilde{k}(x_i, x_j)$$

Claim

The nonlinear PCs can be computed directly in terms of the matrix $\tilde{K}$.

- It can be shown that:
$$\tilde{K} = \left(I - \frac{1}{n}11^T\right)K\left(I - \frac{1}{n}11^T\right) = JKJ$$
  - $1 = (1, 1, \ldots, 1)^T \in \mathcal{R}^n$

- We also define the vectors:
$$k_x = [k(x_1, x), \dots, k(x_n, x)]^T$$
$$\tilde{k_x} = \Phi^T(\phi(x) - \overline{\phi}) = [\tilde{k}(x_1, x), \dots, \tilde{k}(x_n, x)]^T$$
- The nonlinear PCs can then be computed as:
$$y_i = w_i^T \Phi^T \big(\phi(x) - \overline{\phi}\big) = w_i^T \tilde{k}_x, \;\; i = 1, 2, \dots, d$$
  - w_i are the **eignevectors** of $\tilde{K}$:
$$\tilde{K} w_i = \lambda_i w_i$$
  - V_d and Λ_d are obtained from the top-d eigenvectors of $\tilde{K}$'s eigen-decomposition:
$$\tilde{K} = V_{\tilde{K}} \Lambda_{\tilde{K}} V_{\tilde{K}}^T$$
  - Note that if $\tilde{K}$ is a valid kernel, then $\tilde{K} = \tilde{\Phi}^T \tilde{\Phi}$ for some mapping $\tilde{\Phi}$. In this case: $\tilde{\Phi}^T = V_{\tilde{K}} \Lambda^{\frac{1}{2}}$.
  - Concatenating the components y_i we get (the projection):
$$Y = (y_1, y_2, \dots, y_d)^T = \Lambda_d^{-1} W_d^T \Phi^T \Phi = \Lambda_d^{-1} W_d^T \tilde{K}$$

- The low-dimensional nonlinear representation is obtained from the top-d eigenvectors of the matrix $\tilde{K}$.
  - The problem has been solved in the high-dimensional feature space ϕ while not explicitly computing ϕ or even working in $\mathcal{R}^m$.

Kernel-PCA Algorithm

- **Input**: A set of points $\mathcal{X} = \{x_1, x_2, \dots, x_N\} \subset \mathcal{R}^D$, and a mapping $\phi : \mathcal{R}^D \to \mathcal{R}^M$ **or** a symmetric positive definite kernel function $k : \mathcal{R}^D \times \mathcal{R}^D \to \mathcal{R}$

1. Compute $\overline{\phi} = \frac{1}{N} \sum_j \phi(x_j)$ and the centered embedded data matrix Φ **or** the kernel matrix K.
2. Compute the centered kernel *matrix*:
$$\tilde{\mathcal{K}} = \Phi^T \Phi \text{ or } \tilde{\mathcal{K}} = \big(I - \frac{1}{n} 1 1^T\big) K \big(I - \frac{1}{n} 1 1^T\big) = JKJ \text{ or } \big(\tilde{k}(x_i, x_j)\big) \in \mathcal{R}^{N \times N}$$
3. Compute the eigenvectors $w_i \in \mathcal{R}^N$ of $\tilde{\mathcal{K}}$:
$$\tilde{\mathcal{K}} w_i = \lambda_i w_i.$$
   - For every data point x, its i^{th} nonlinear principal component is given by:
$$y_i = \lambda_i^{-1} w_i^T \Phi^T (\phi(x) - \overline{\phi}) = \lambda_i^{-1} w_i^T [\tilde{k}(x_1, x), \dots, \tilde{k}(x_N, x)]^T$$
   for $i = 1, 2, \dots, d$
4. The projection is given by:
$$Y = (y_1, y_2, \dots, y_d)^T = \Lambda_d^{-1} W_d^T \Phi^T \Phi = \Lambda_d^{-1} W_d^T \tilde{K}$$

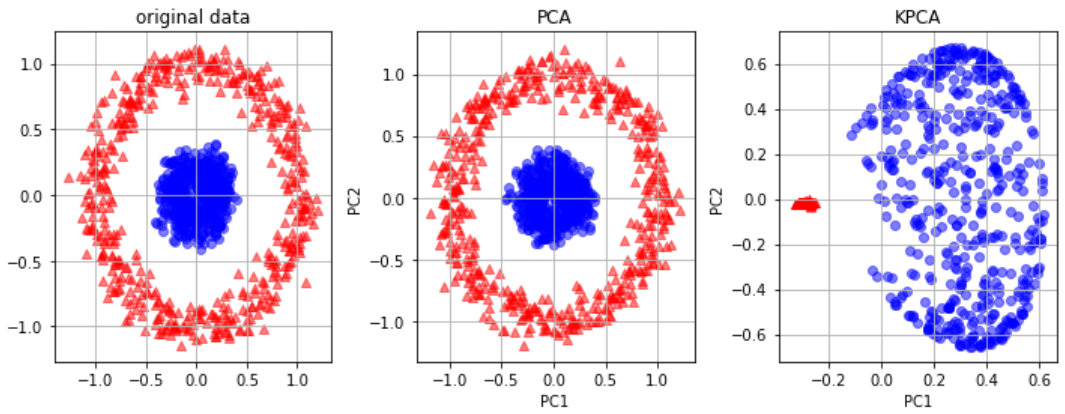
- **Output**: A set of points $\{y_j\}_{j=1}^N$ lying in $\mathcal{R}^d$, where y_{ij} in the i^{th} nonlinear principal component of x_j for $i = 1, \dots, d$ and $j = 1, \dots, N$

Important Notes

1. When using KPCA, you must save the data in memory since you need the kernel matrix to compute the projection.
2. When using KPCA and want to know the mapping of a new unseen data point (not part of the N data points), you have to re-calcualte the kernel between all of your data and the new point.

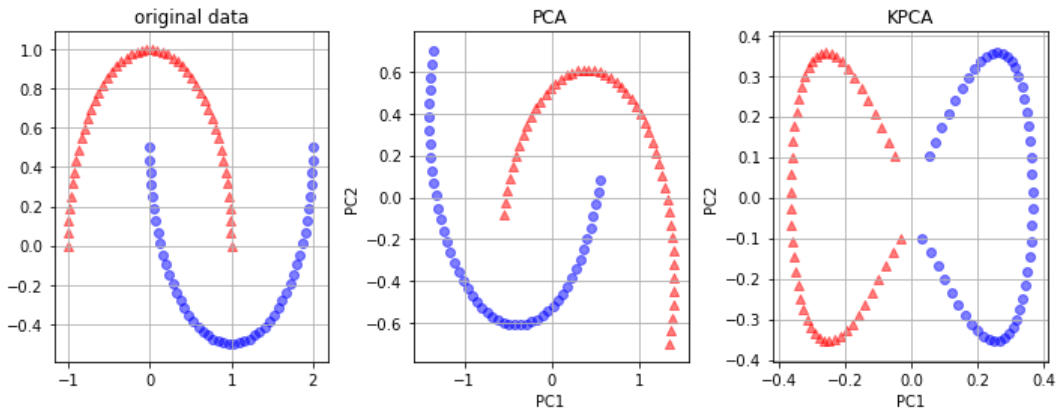
In [4]:
```
def plot_circles_example():
    # example - circles
    X, y = make_circles(n_samples=1000, random_state=123, noise=0.1, factor=0.2)
    pca = PCA(n_components=2)
    kpca = KernelPCA(n_components=2, kernel='rbf', gamma=15)  # Radial Basis Function kernel
    X_pca = pca.fit_transform(X)
    X_kpca = kpca.fit_transform(X)

    # plot
    fig = plt.figure(figsize=(10,4))

    # original data
    ax1 = fig.add_subplot(1,3,1)
    ax1.scatter(X[y==0, 0], X[y==0, 1], color='red', marker='^', alpha=0.5)
    ax1.scatter(X[y==1, 0], X[y==1, 1], color='blue', marker='o', alpha=0.5)
    ax1.grid()
    ax1.set_title("original data")

    # pca
    ax2 = fig.add_subplot(1,3,2)
    ax2.scatter(X_pca[y==0, 0], X_pca[y==0, 1], color='red', marker='^', alpha=0.5)
    ax2.scatter(X_pca[y==1, 0], X_pca[y==1, 1], color='blue', marker='o', alpha=0.5)
    ax2.grid()
    ax2.set_title("PCA")
    ax2.set_xlabel("PC1")
    ax2.set_ylabel("PC2")

    # kpca
    ax3 = fig.add_subplot(1,3,3)
    ax3.scatter(X_kpca[y==0, 0], X_kpca[y==0, 1], color='red', marker='^', alpha=0.5)
    ax3.scatter(X_kpca[y==1, 0], X_kpca[y==1, 1], color='blue', marker='o', alpha=0.5)
    ax3.grid()
    ax3.set_title("KPCA")
    ax3.set_xlabel("PC1")
    ax3.set_ylabel("PC2")

    plt.tight_layout()
```

In [5]:
```
plot_circles_example()
```



In [6]:
```
def plot_moons_example():
    # example - moons
    X, y = make_moons(n_samples=100, random_state=123)
    pca = PCA(n_components=2)
    kpca = KernelPCA(n_components=2, kernel='rbf', gamma=15)  # Radial Basis Function kernel
    X_pca = pca.fit_transform(X)
    X_kpca = kpca.fit_transform(X)

    # plot
    fig = plt.figure(figsize=(10,4))

    # original data
    ax1 = fig.add_subplot(1,3,1)
    ax1.scatter(X[y==0, 0], X[y==0, 1], color='red', marker='^', alpha=0.5)
    ax1.scatter(X[y==1, 0], X[y==1, 1], color='blue', marker='o', alpha=0.5)
    ax1.grid()
    ax1.set_title("original data")

    # pca
    ax2 = fig.add_subplot(1,3,2)
    ax2.scatter(X_pca[y==0, 0], X_pca[y==0, 1], color='red', marker='^', alpha=0.5)
    ax2.scatter(X_pca[y==1, 0], X_pca[y==1, 1], color='blue', marker='o', alpha=0.5)
    ax2.grid()
    ax2.set_title("PCA")
    ax2.set_xlabel("PC1")
    ax2.set_ylabel("PC2")

    # kpca
    ax3 = fig.add_subplot(1,3,3)
    ax3.scatter(X_kpca[y==0, 0], X_kpca[y==0, 1], color='red', marker='^', alpha=0.5)
    ax3.scatter(X_kpca[y==1, 0], X_kpca[y==1, 1], color='blue', marker='o', alpha=0.5)
    ax3.grid()
    ax3.set_title("KPCA")
    ax3.set_xlabel("PC1")
    ax3.set_ylabel("PC2")

    plt.tight_layout()
```

In [7]:
```
plot_moons_example()
```



Recommended Videos

Warning!

- These videos do not replace the lectures and tutorials.
- Please use these to get a better understanding of the material, and not as an alternative to the written material.

Video By Subject

- PCA (1) - StatQuest: Principal Component Analysis (PCA), Step-by-Step (https://www.youtube.com/watch?v=FgakZw6K1QQ)
- PCA (2) - Principal Component Analysis (PCA) - Computerphile (https://www.youtube.com/watch?v=TJdH6rPA-TI)
- The Kernel Trick - Machine Learning Fundamentals - The Kernel Trick (https://www.youtube.com/watch?v=JiM_LXpAtLc)
- Kernel PCA - David Thompson: Nonlinear Dimensionality Reduction: KPCA (https://www.youtube.com/watch?v=HbDHohXPLnU)

Credits

- Icons from Icon8.com (https://icons8.com/) - https://icons8.com (https://icons8.com)
- Datasets from Kaggle (https://www.kaggle.com/) - https://www.kaggle.com/ (https://www.kaggle.com/)