02_the_math_behind_pca

September 29, 2021

1 How the PCA algorithm works

PCA makes several assumptions that are important to keep in mind. These include: - high variance implies a high signal-to-noise ratio - the data is standardized so that the variance is comparable across features - linear transformations capture the relevant aspects of the data, and - higher-order statistics beyond the first and second moment do not matter, which implies that the data has a normal distribution

The emphasis on the first and second moments align with standard risk/return metrics, but the normality assumption may conflict with the characteristics of market data.

The algorithm finds vectors to create a hyperplane of target dimensionality that minimizes the reconstruction error, measured as the sum of the squared distances of the data points to the plane. As illustrated above, this goal corresponds to finding a sequence of vectors that align with directions of maximum retained variance given the other components while ensuring all principal component are mutually orthogonal.

In practice, the algorithm solves the problem either by computing the eigenvectors of the covariance matrix or using the singular value decomposition.

1.1 Imports & Settings

[1]:
```
%matplotlib inline
import pandas as pd
import numpy as np
from numpy.linalg import inv, eig, svd
from numpy.random import uniform, randn, seed
from itertools import repeat
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from sklearn.decomposition import PCA
from sklearn.datasets import load_digits

pd.options.display.float_format = '{:,.2f}'.format
seed(42)
```

[2]:
```
def format3D(axis, labels=('x', 'y', 'z'), limits=None):
    """3D plot helper function to set labels, pane color, and axis limits"""
    axis.set_xlabel('\n${}$'.format(labels[0]), linespacing=3)
```

```
    axis.set_ylabel('\n${}$'.format(labels[1]), linespacing=3)
    axis.set_zlabel('\n${}$'.format(labels[2]), linespacing=3)
    transparent = (1.0, 1.0, 1.0, 0.0)
    axis.w_xaxis.set_pane_color(transparent)
    axis.w_yaxis.set_pane_color(transparent)
    axis.w_zaxis.set_pane_color(transparent)
    if limits:
        axis.set_xlim(limits[0])
        axis.set_ylim(limits[1])
        axis.set_zlim(limits[2])
```

1.2 Create a noisy 3D Ellipse

We illustrate the computation using a randomly generated three-dimensional ellipse with 100 data points.

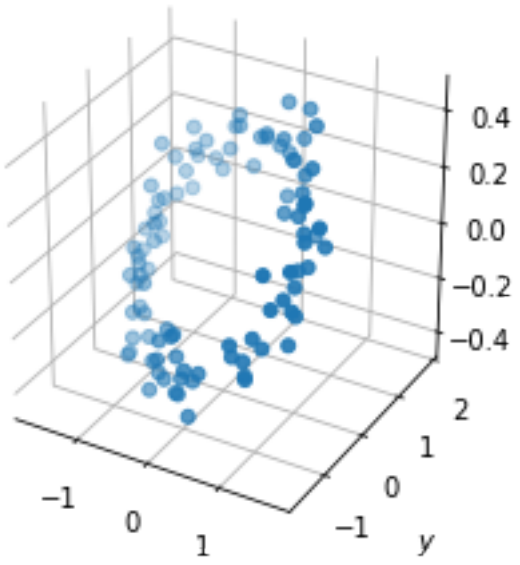
[3]:
```
n_points, noise = 100, 0.1
angles = uniform(low=-np.pi, high=np.pi, size=n_points)
x = 2 * np.cos(angles) + noise * randn(n_points)
y = np.sin(angles) + noise * randn(n_points)

theta = np.pi/4 # 45 degree rotation
rotation_matrix = np.array([[np.cos(theta), -np.sin(theta)],
                            [np.sin(theta), np.cos(theta)]])

rotated = np.column_stack((x, y)).dot(rotation_matrix)
x, y = rotated[:, 0], rotated[:, 1]

z = .2 * x  + .2 * y + noise * randn(n_points)
data = np.vstack((x, y, z)).T
```

[4]:
```
data.shape
```

[4]:
```
(100, 3)
```

1.2.1 Plot the result

[5]:
```
ax = plt.figure().gca(projection='3d')
ax.set_aspect('equal')
ax.scatter(x, y, z, s=25)
format3D(ax)
# plt.gcf().set_size_inches(12,12)
# plt.tight_layout();
```



1.3 Principal Components using scikit-learn

The sklearn.decomposition.PCA implementation follows the standard API based on fit() and transform() methods that compute the desired number of principal components and project the data into the component space, respectively. The convenience method fit_transform() accomplishes this in a single step.

PCA offers three different algorithms that can be specified using the svd_solver parameter: - full computes the exact SVD using the LAPACK solver provided by scipy, - arpack runs a truncated version suitable for computing less than the full number of components. - randomized uses a sampling-based algorithm that is more efficient when the data set has more than 500 observations and features, and the goal is to compute less than 80% of the components - auto also randomized where most efficient, otherwise, it uses the full SVD.

Other key configuration parameters of the PCA object are: - n_components: compute all principal components by passing None (the default), or limit the number to int. For svd_solver=full, there are two -additional options: a float in the interval [0, 1] computes the number of components required to retain the corresponding share of the variance in the data, and the option mle estimates the number of dimensions using maximum likelihood. - whiten: if True, it standardizes the component vectors to unit variance that in some cases can be useful for use in a predictive model (default is False)

[6]:
```
pca = PCA()
pca.fit(data)
C = pca.components_.T # columns = principal components
C
```

[6]:
```
array([[ 0.71409739,  0.66929454,  0.20520656],
       [-0.70000234,  0.68597301,  0.1985894 ],
       [ 0.00785136,  0.28545725, -0.95835928]])
```

1.3.1 First principal component

[7]:
```
C[:, 0]
```

[7]:
```
array([ 0.71409739, -0.70000234,  0.00785136])
```

1.3.2 Explained Variance

[8]:
```
explained_variance = pca.explained_variance_
explained_variance
```

[8]:
```
array([1.92923132, 0.55811089, 0.00581353])
```

[9]:
```
np.allclose(explained_variance/np.sum(explained_variance),
            pca.explained_variance_ratio_)
```

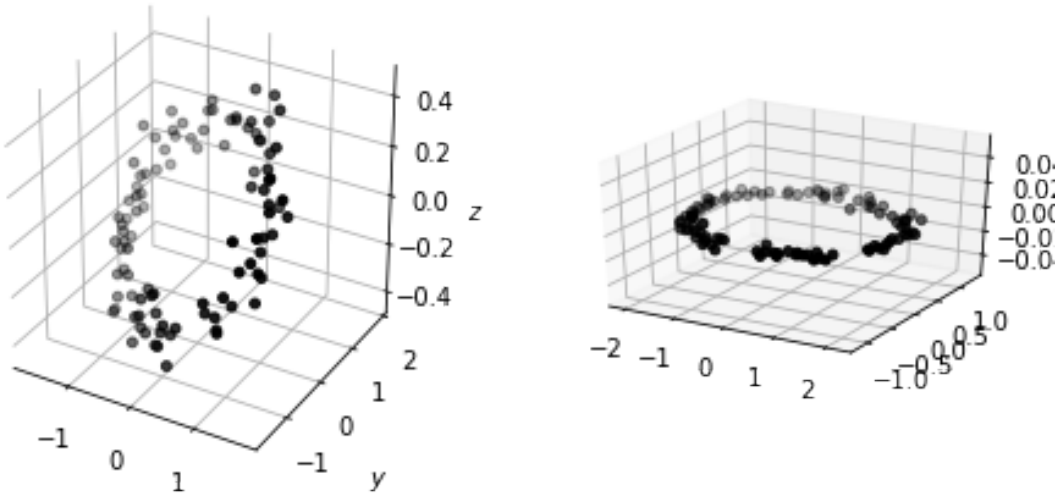
[9]:
```
True
```

1.3.3 PCA to reduce dimensions

[10]:
```
pca2 = PCA(n_components=2)
projected_data  = pca2.fit_transform(data)
projected_data.shape
```

[10]:
```
(100, 2)
```

[13]:
```
fig = plt.figure(figsize=plt.figaspect(0.5))
ax1 = fig.add_subplot(1, 2, 1, projection='3d')
ax1.set_aspect('equal')
ax1.scatter(x, y, z, s=15, c='k')
format3D(ax1)

ax2 = fig.add_subplot(1, 2, 2, projection='3d')
ax2.set_aspect('equal')
ax2.scatter(*projected_data.T, s=15, c='k')
format3D(ax1)
```



[14]:
```
pca2.explained_variance_ratio_
```

[14]:
```
array([0.77381099, 0.22385721])
```

1.4 Principal Components from Covariance Matrix

We first compute the principal components using the square covariance matrix with the pairwise sample covariances for the features xi, xj, i, j = 1, ..., n as entries in row i and column j:

$$cov_{i,j} = \frac{\sum_{k=1}^{N}(x_{ik}-\bar{x_i})(x_{jk}-\bar{x_j})}{N-1}$$

Using `numpy`, we implement this as follows, where the pandas `DataFrame` data contains the 100 data points of the ellipse:

[15]:
```
cov = np.cov(data.T) # each row represents a feature
cov.shape
```

[15]:
```
(3, 3)
```

1.4.1 Eigendecomposition: Eigenvectors & Eigenvalues

- The Eigenvectors w_i and Eigenvalues λ_i for a square matrix M are defined as follows:

$$Mw_i = \lambda_i w_i$$

- This implies we can represent the matrix M using Eigenvectors and Eigenvalues, where W is a matrix that contains the Eigenvectors as column vectors, and L is a matrix that contains the λ_i as diagonal entries (and 0s otherwise):

$$M = WLW^{-1}$$

Next, we calculate the eigenvectors and eigenvalues of the covariance matrix. The eigenvectors contain the principal components (where the sign is arbitrary):

[16]:
```
eigen_values, eigen_vectors = eig(cov)
```

1.4.2 eigenvectors = principal components

[17]:
```
eigen_vectors
```

[17]:
```
array([[ 0.71409739, -0.66929454, -0.20520656],
       [-0.70000234, -0.68597301, -0.1985894 ],
       [ 0.00785136, -0.28545725,  0.95835928]])
```

We can compare the result with the result obtained from sklearn and find that they match in absolute terms:

[18]:
```
np.allclose(np.abs(C), np.abs(eigen_vectors))
```

[18]:
```
True
```

eigenvalues = explained variance

[19]:
```
eigen_values
```

[19]:
```
array([1.92923132, 0.55811089, 0.00581353])
```

[20]:
```
np.allclose(explained_variance, eigen_values)
```

[20]:
```
True
```

Check that Eigendecomposition works We can also verify the eigendecomposition, starting with the diagonal matrix L that contains the eigenvalues:

[21]:
```
ev = np.zeros((3, 3))
np.fill_diagonal(ev, eigen_values)
ev # diagonal matrix
```

[21]:
```
array([[1.92923132, 0.        , 0.        ],
       [0.        , 0.55811089, 0.        ],
       [0.        , 0.        , 0.00581353]])
```

We find that the result does indeed hold:

[22]:
```
decomposition = eigen_vectors.dot(ev).dot(inv(eigen_vectors))
np.allclose(cov, decomposition)
```

[22]:
```
True
```

1.4.3 Preferred: Singular Value Decomposition

Next, we'll look at the alternative computation using the Singular Value Decomposition (SVD). This algorithm is slower when the number of observations is greater than the number of features (the typical case) but yields better numerical stability, especially when some of the features are strongly correlated (often the reason to use PCA in the first place).

SVD generalizes the eigendecomposition that we just applied to the square and symmetric covariance matrix to the more general case of m x n rectangular matrices. It has the form shown at the center of the following figure. The diagonal values of Σ are the singular values, and the transpose of V* contains the principal components as column vectors.

Requires centering your data! In this case, we need to make sure our data is centered with mean zero (the computation of the covariance above took care of this):

[23]:
```
n_features = data.shape[1]
data_ = data - data.mean(axis=0)
```

We find that the decomposition does indeed reproduce the standardized data:

[24]:
```
cov_manual = data_.T.dot(data_)/(len(data)-1)
np.allclose(cov_manual, cov)
```

[24]:
```
True
```

[25]:
```
U, s, Vt = svd(data_)
U.shape, s.shape, Vt.shape
```

[25]:
```
((100, 100), (3,), (3, 3))
```

[26]:
```
# Convert s from vector to diagonal matrix
S = np.zeros_like(data_)
S[:n_features, :n_features] = np.diag(s)
S.shape
```

[26]:
```
(100, 3)
```

Show that the result indeed decomposes the original data

[27]:
```
np.allclose(data_, U.dot(S).dot(Vt))
```

[27]:
```
True
```

Confirm that V^T contains the principal components

[28]:
```
np.allclose(np.abs(C), np.abs(Vt.T))
```

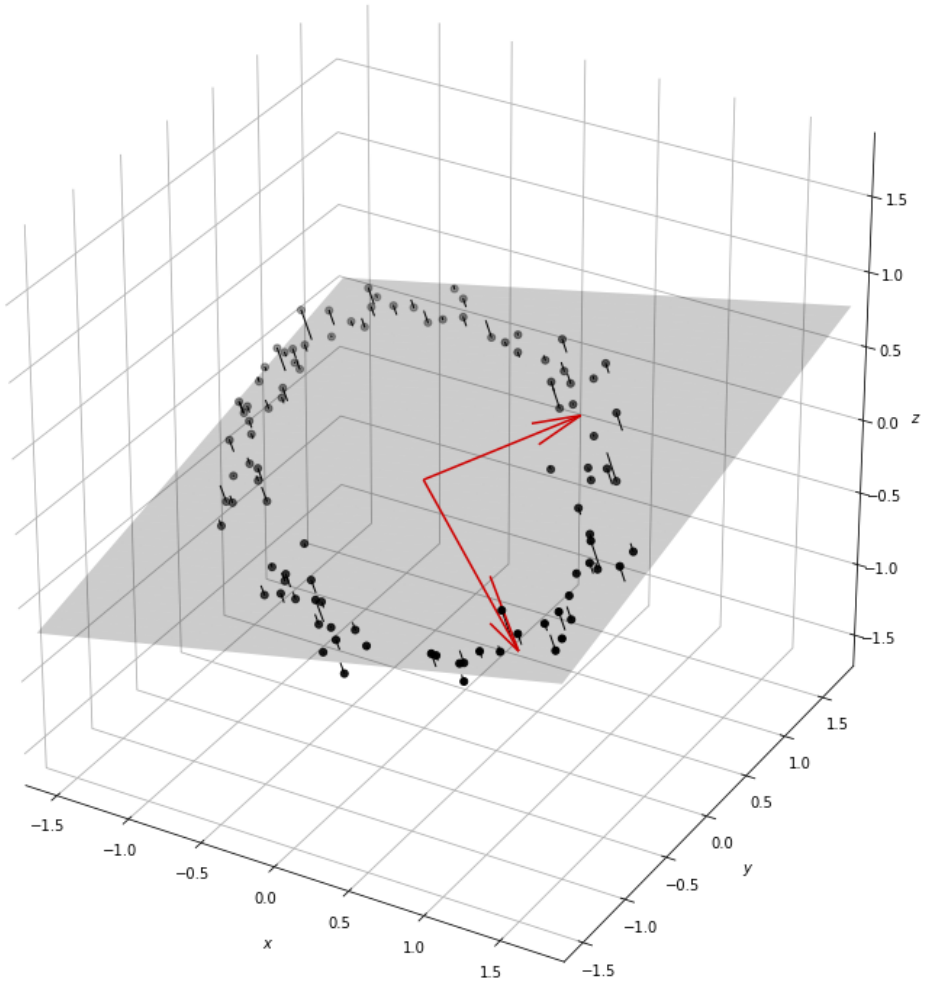
[28]:
```
True
```

1.4.4 Visualize 2D Projection

[29]:
```
pca = PCA(n_components=2)
data_2D = pca.fit_transform(data)

min_, max_ = data[:, :2].min(), data[:, :2].max()
xs, ys = np.meshgrid(np.linspace(min_,max_, n_points),
                     np.linspace(min_,max_, n_points))

normal_vector = np.cross(pca.components_[0], pca.components_[1])
d = -pca.mean_.dot(normal_vector)
zs = (-normal_vector[0] * xs - normal_vector[1] * ys - d) * 1 / normal_vector[2]
```

[30]:
```
C = pca.components_.T.copy()
proj_matrix = C.dot(inv(C.T.dot(C))).dot(C.T)
C[:,0] *= 2
```

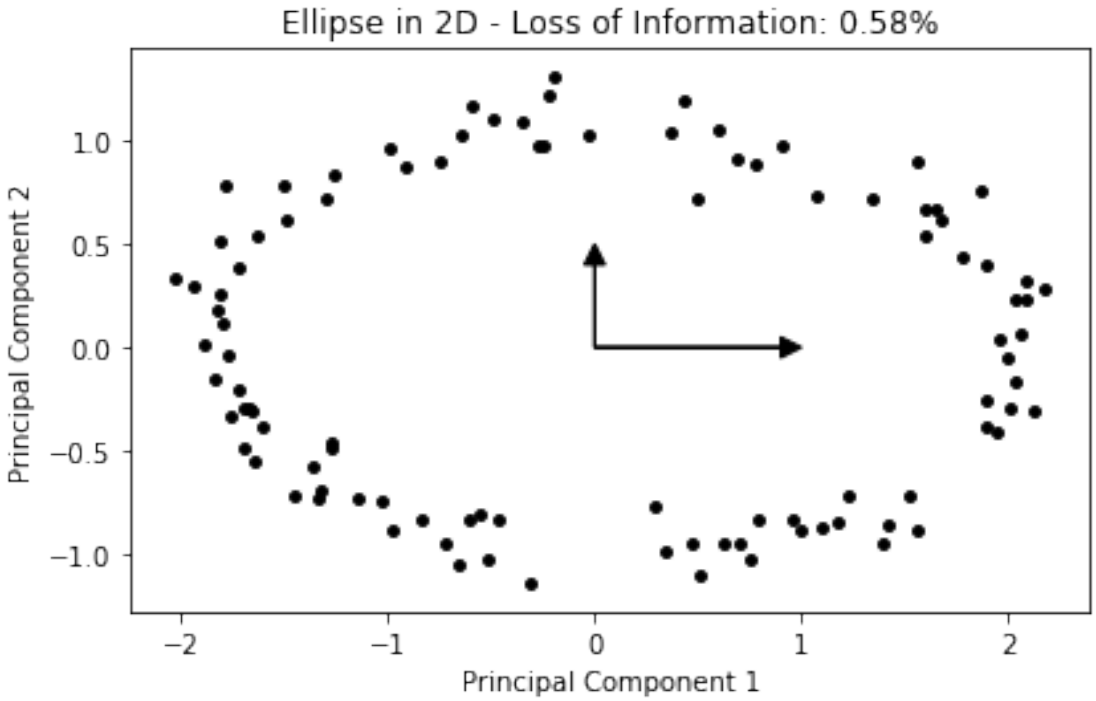
[32]:
```
ax = plt.figure(figsize=(14,14)).gca(projection='3d')
ax.set_aspect('equal')

ax.plot_surface(xs, ys, zs, alpha=0.2, color='black',
                linewidth=1, antialiased=True)
ax.scatter(x, y, z, c='k', s=25)

for i in range(n_points):
    ax.plot(*zip(proj_matrix.dot(data[i]), data[i]),
            color='k', lw=1)

origin = np.zeros((2, 3))
X, Y, Z, U, V, W = zip(*np.hstack((origin, C.T)))
ax.quiver(X, Y, Z, U, V, W, color='red')

format3D(ax, limits=list(repeat((min_, max_), 3)))
```



1.4.5 2D Representation

[33]:
```
data_3D_inv = pca.inverse_transform(data_2D)
avg_error = np.mean(np.sum(np.square(data_3D_inv-data), axis=1))
fig = plt.figure()
ax = fig.add_subplot(111, aspect='equal',
     xlabel='Principal Component 1',
     ylabel='Principal Component 2',
     title='Ellipse in 2D - Loss of Information: {:.2%}'.format(avg_error))

ax.scatter(data_2D[:, 0], data_2D[:, 1], color='k', s=15)
ax.arrow(0, 0, 0, .5, head_width=0.1, length_includes_head=True,
         head_length=0.1, fc='k', ec='k')
```

```
ax.arrow(0, 0, 1, 0, head_width=0.1, length_includes_head=True,
         head_length=0.1, fc='k', ec='k')
fig.tight_layout();
```



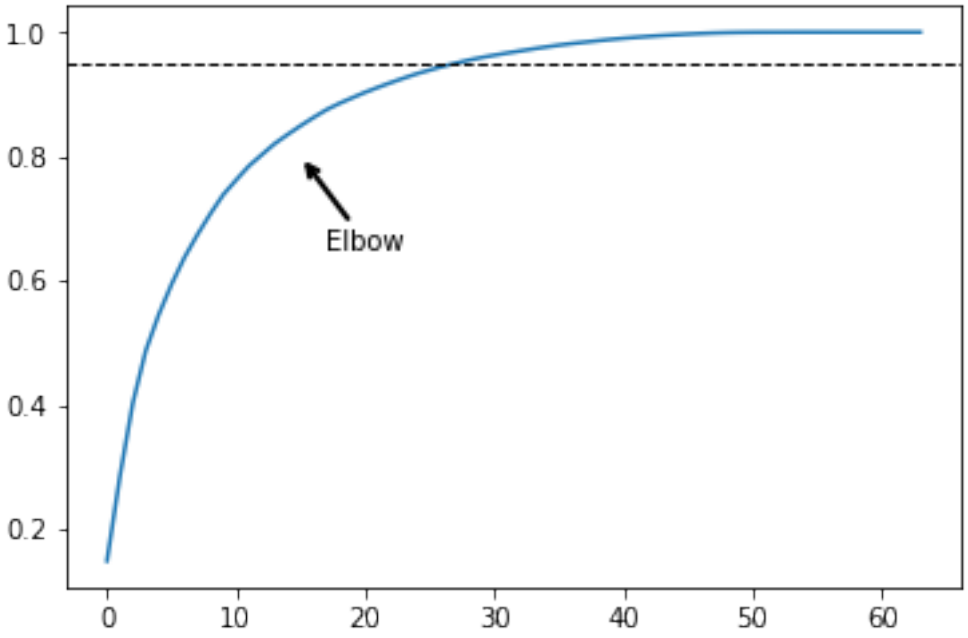
1.4.6 How many Components to represent 64 dimensions?

[34]:
```
n_classes = 10
digits = load_digits(n_class=n_classes)
X = digits.data
y = digits.target
n_samples, n_features = X.shape
n_samples, n_features
```

[34]: (1797, 64)

Evaluate the cumulative explained variance

[35]:
```
pca = PCA(n_components=64).fit(X)
pd.Series(pca.explained_variance_ratio_).cumsum().plot()
plt.annotate('Elbow', xy=(15, .8), xycoords='data', xytext=(20, .65),
    textcoords='data', horizontalalignment='center',
    arrowprops=dict(color='k', lw=2, arrowstyle="->")
)
plt.axhline(.95, c='k', lw=1, ls='--');
```



1.4.7 Automate generation of Components

[36]:
```
pca = PCA(n_components=.95).fit(X)
pca.components_.shape
```

[36]: (29, 64)

[ ]:
```

```