DM561 – Linear Algebra with Applications / DM562 – Scientific Computing

Week 50, 2020

Principal Component Analysis and Eigenfaces

Daniel Merkle

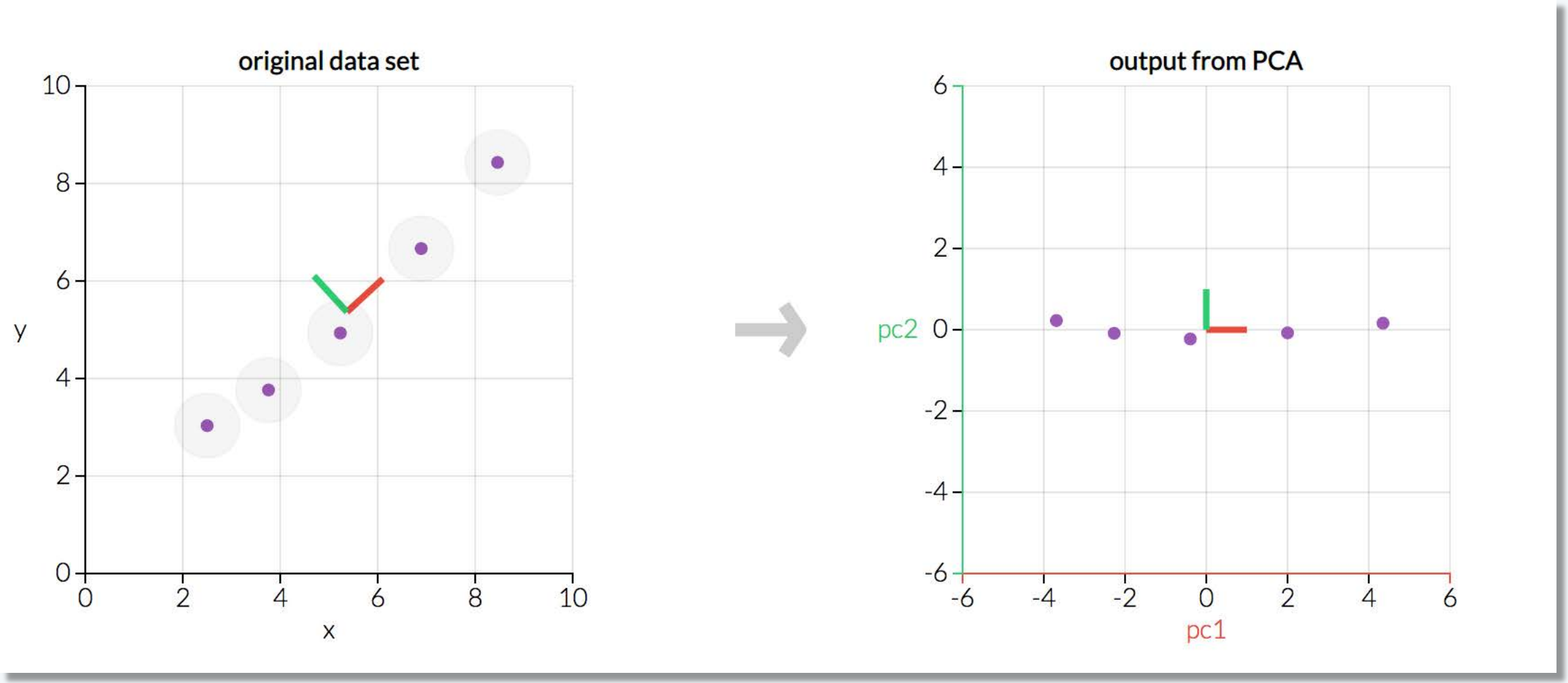
daniel@imada.sdu.dk

Introduction and Motivational Example

- PCA is considered the "the mother of all methods in multivariate data analysis"
- Dimensionality reduction
- Exploit linear dependencies in data
- PCA can be seen as a method to compute a **new coordinate system** formed by latent* variables, which is **orthogonal**, and where only the **most informative dimensions** are used.
- Dimension reduction by PCA is mainly used for
  - Visualization of multivariate data
  - Transformation of highly correlating variables into a smaller set of uncorrelated
  - Separation of relevant information from noise
  - Combination of several variables that characterize a chemical-technological process

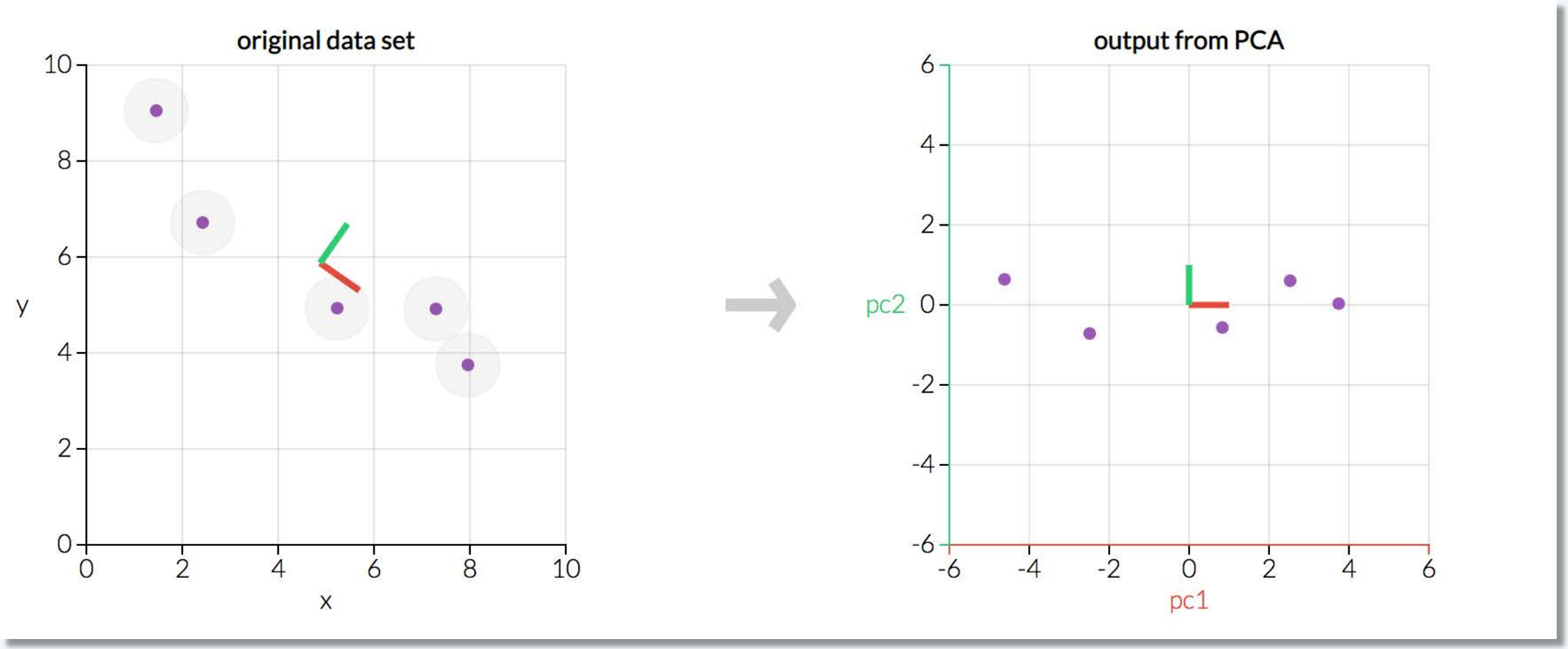
*variables that are not directly observed but are rather inferred (through a mathematical model)

PCA finds a new coordinate system



Nice visualization : http://setosa.io/ev/principal-component-analysis/

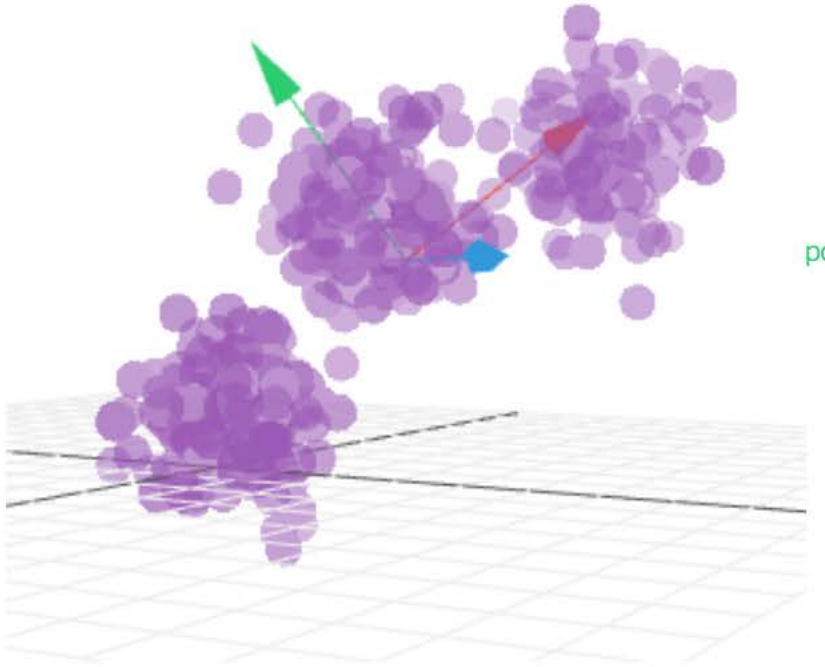
New Dimensions are called Principal Components



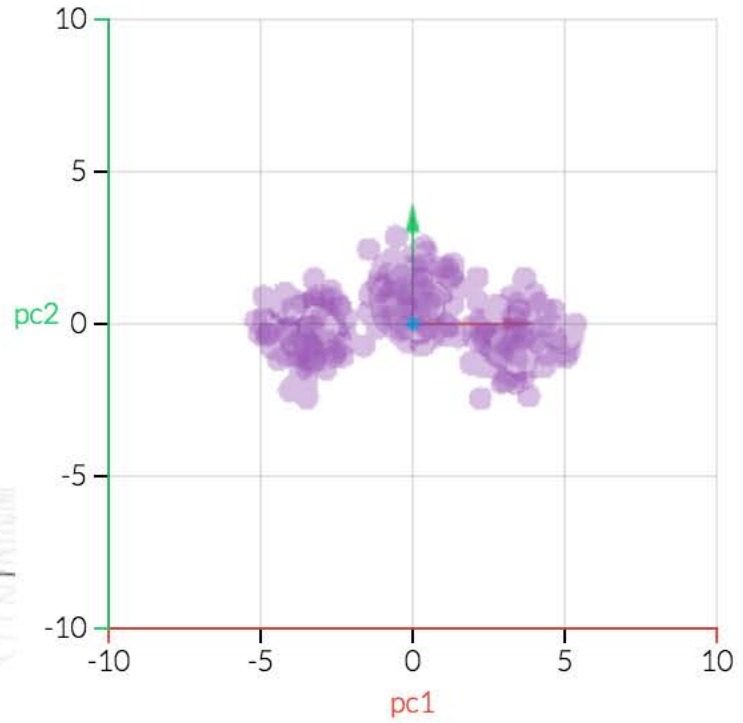
This transformation is defined in such a way that the **first principal component has the largest possible variance** (that is, accounts for as much of the variability in the data as possible), and **each succeeding** component in turn has the highest variance possible under the constraint that it is **orthogonal** to the preceding components.

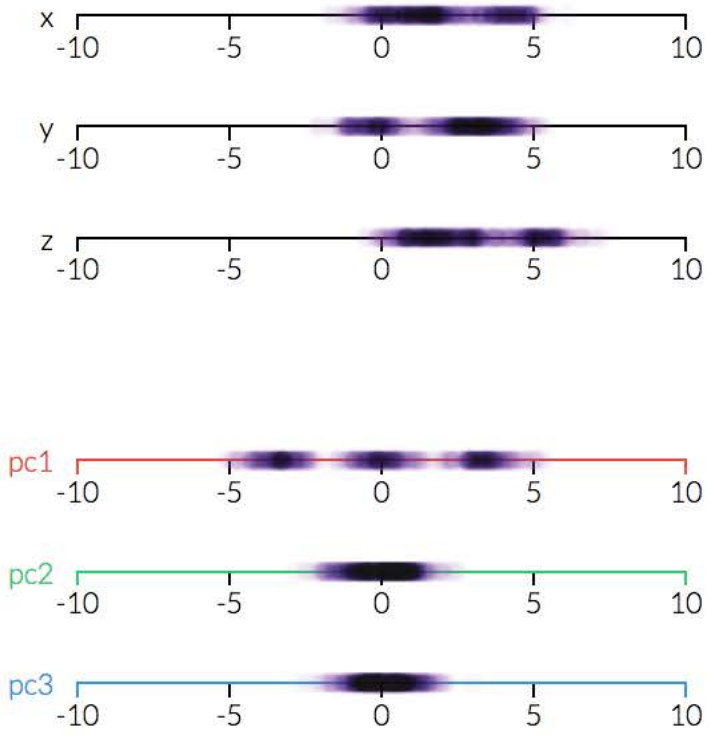
Dimensionality Reduction

Original (observed) data



Projected data





Mathematical Background

- Orthogonal Basis
- Linear Transformation
- Covariance Matrix
- And (we will use without proof):
  - The inverse of an orthonormal matrix is its transpose (1)
  - For any matrix A, A^TA and AA^T are symmetric.
  - A matrix is symmetric if and only if it is orthogonally diagonalizable.
  - A symmetric matrix is diagonalized by a matrix of its orthonormal eigenvectors, i.e., for any squared symmetric matrix A, there exists a diagonal matrix D, such that $A = E\ D\ E^T$,where the i-th column of E is the i-th eigenvector of E (special case of an **SVD decomposition**). The orthonormal column vectors of E span a space, and due to (1) $E^{-1} = E^T$

Change of Basis – Linear Transformation

[Presented on (virtual) blackboard. More details : page 3 of *A Tutorial on Principal Component Analysis (*Jonathon Shlens)]

Original data: X (m x n matrix)

P is the matrix that transforms X into Y

Y (m x n matrix) is the new representation of the data set

$$Y = PX$$

The rows of P are the new basis vectors

Later, m will be the number of features/dimensions, n will be the number of samples

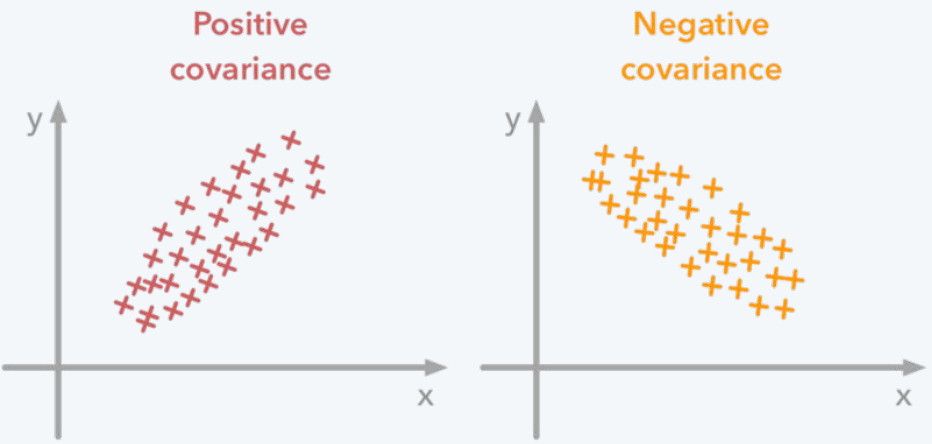
Mathematical Prerequisites - Covariance

$$A = \{a_1, \ldots, a_n\} \qquad B = \{b_1, \ldots, b_n\}$$

$$\operatorname{cov}(A, B) = \sigma^2_{AB} = \frac{1}{n} \sum_{i=1}^{n} (a_i - \bar{a})\left(b_i - \bar{b}\right)$$

Assuming A and B are mean centered:

$$\sigma^2_{AB} = \frac{1}{n} \sum_{i=1}^{n} a_i b_i$$



Assuming (mean-centered) row vectors for a and b:

$$\mathbf{a} = (a_1, \ldots, a_n)$$
$$\mathbf{b} = (b_1, \ldots, b_n)$$
$$\sigma^2_{\mathbf{ab}} = \frac{1}{n}\mathbf{ab}$$

[Bessel's correction: n-1 in the denominator, see https://en.wikipedia.org/wiki/Bessel%27s_correction]

Covariance Matrix

Assume data (m types of measurements) given as

$$X = \begin{bmatrix} \mathbf{x_1} \\ \vdots \\ \mathbf{x_m} \end{bmatrix}$$

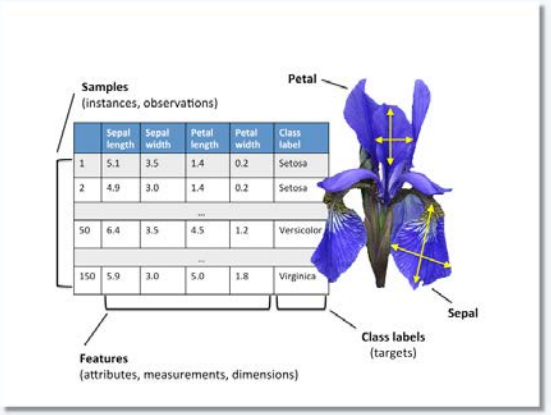
A note of caution, sometimes the role of column and row (type of measurement/samples) is switched. Obviously, then:

$$X = [\mathbf{x_1} \ldots \mathbf{x_m}]$$

$$C_X = \frac{1}{n} X^T X$$

	sepal_len	sepal_wid	petal_len	petal_wid	class
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

Data given as a matrix of size
samples x features
m x n
150 x 4



Each ***row*** of X corresponds to all m measurements of a particular type. Each ***column*** of X corresponds to a set of all n measurements from one particular trial.

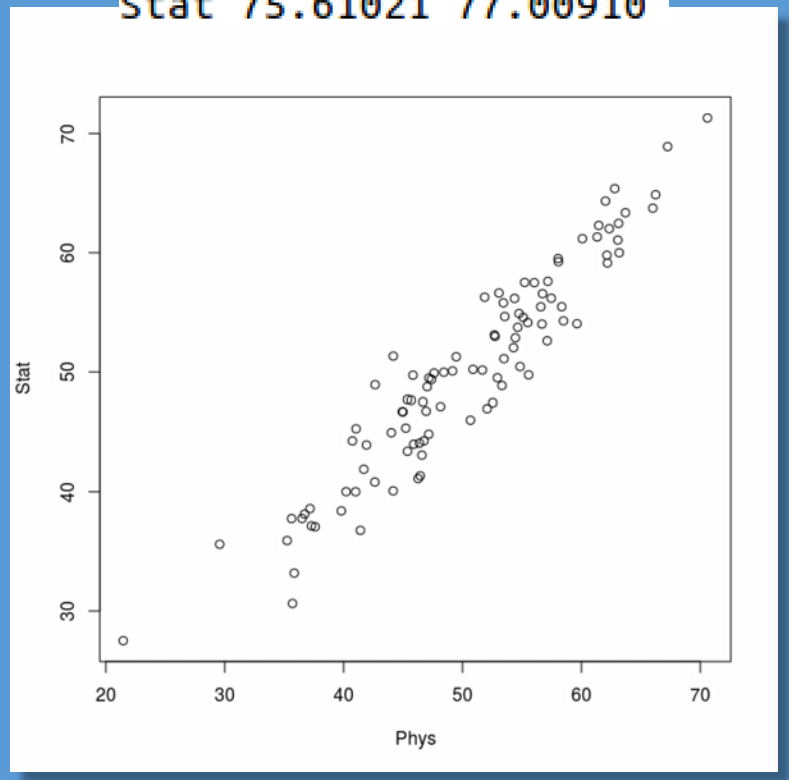
Covariance Matrix : $C_X = \frac{1}{n} X X^T$

captures the covariance between all possible pairs of measurements. The covariance values reflect the noise and redundancy in our measurements.

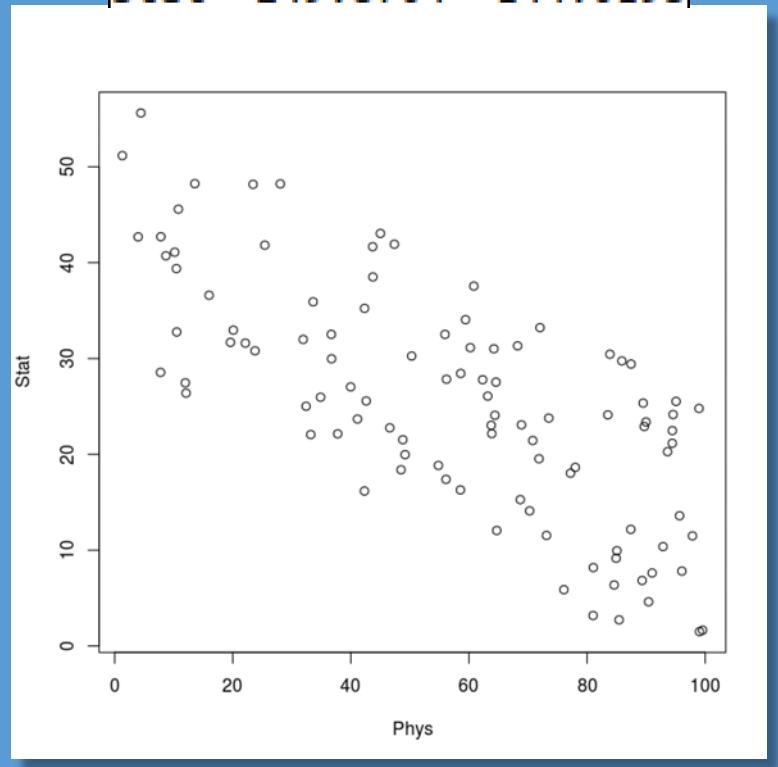
Properties of the Covariance Matrix

- C_X is a square symmetric $m \times m$ matrix (m is the number of features)
- The diagonal terms of C_X are the *variance* of particular measurement types.
- The off-diagonal terms of C_X are the *covariance* between measurement types.

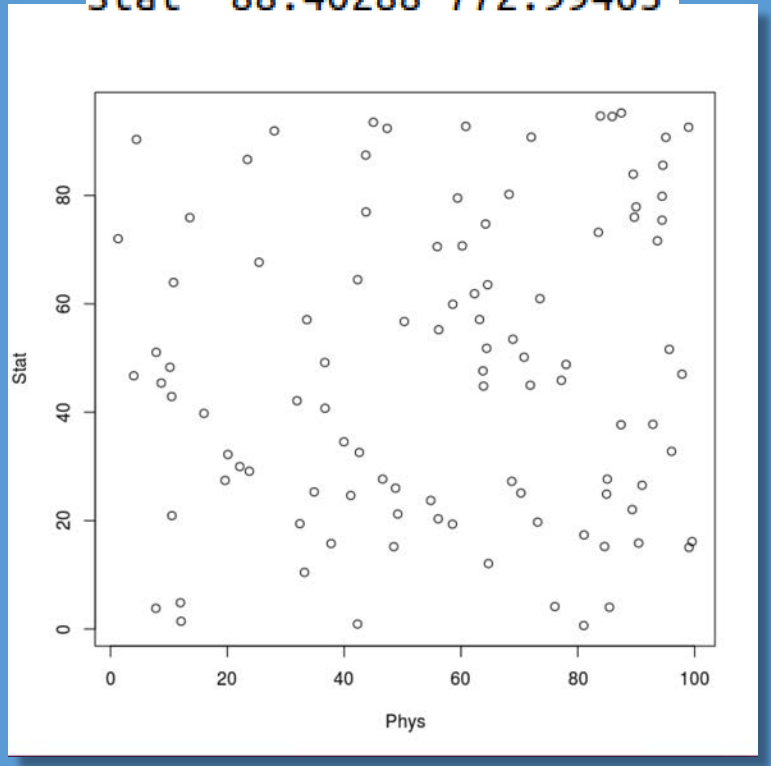
	Phys	Stat
Phys	82.24725	75.61021
Stat	75.61021	77.00910



	Phys	Stat
Phys	829.1742	-249.8704
Stat	-249.8704	144.0193



	Phys	Stat
Phys	829.17425	88.40288
Stat	88.40288	772.99465

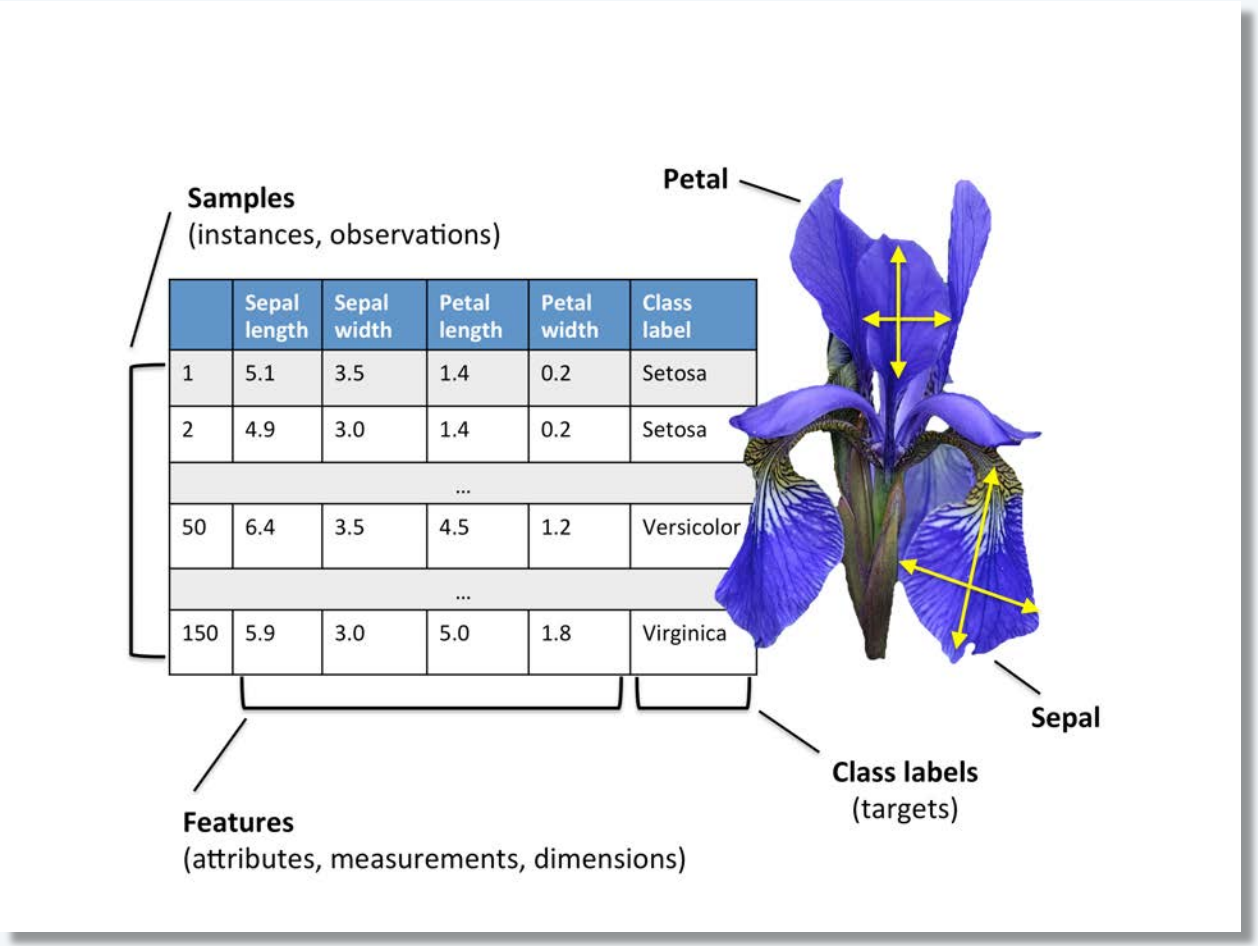


Example: Physics and Statistics grade [0-100 points], measured for 100 students

Iris Example (150 samples, 50 from 3 different species)

https://en.wikipedia.org/wiki/Iris_flower_data_set

	sepal_len	sepal_wid	petal_len	petal_wid	class
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica



Data given as a matrix of size

samples x features

n x m

150 x 4

$$X = [\mathbf{x_1} \dots \mathbf{x_m}]$$

$$C_X = \frac{1}{n} X^T X$$

Question!

Pretend we have the option of manipulating the covariance matrix C_X (by a change of basis such that X will be transformed into Y, i.e., Y = PX). We will define our manipulated covariance matrix C_Y. What features do we ideally want to have in C_Y?

- All off-diagonal terms in C_Y should be zero. Thus, **C_Y must be a diagonal matrix**. Or, said another way, Y is de-correlated.
- Each successive **dimension in Y** should be **rank-ordered** according to variance.

Can we find some orthonormal matrix P in Y = PX such that

$$C_Y = \frac{1}{n} Y Y^T$$

is a diagonal matrix?

YES!

[Presented on blackboard. More details : page 3 of *A Tutorial on Principal Component Analysis (*Jonathon Shlens)]

$$
\begin{aligned}
\mathbf{C_Y} &= \frac{1}{n}\mathbf{Y}\mathbf{Y}^T \\
&= \frac{1}{n}(\mathbf{P}\mathbf{X})(\mathbf{P}\mathbf{X})^T \\
&= \frac{1}{n}\mathbf{P}\mathbf{X}\mathbf{X}^T\mathbf{P}^T \\
&= \mathbf{P}(\frac{1}{n}\mathbf{X}\mathbf{X}^T)\mathbf{P}^T \\
\mathbf{C_Y} &= \mathbf{P}\mathbf{C_X}\mathbf{P}^T
\end{aligned}
$$

We select the matrix P to be a matrix where each row p_i is an eigenvector of $\frac{1}{n}XX^T$, i.e., $P = E^T$

$$
\begin{aligned}
C_Y &= P \cdot C_X \cdot P^T \\
&= P \cdot E \cdot D \cdot E^T \cdot P \\
&= E^T \cdot E \cdot D \cdot E^T \cdot E \\
&= E^{-1} \cdot E \cdot D \cdot E^{-1} \cdot E \\
&= D
\end{aligned}
$$

PCA in Python (numpy only)

```
# principal component analysis
from numpy import array
from numpy import mean
from numpy import cov
from numpy.linalg import eig
# define matrix
A = array([
  [1, 2],
  [3, 4],
  [5, 6]])
print(A)
# column means
M = mean(A.T, axis=1)
# center columns by subtracting column means
C = A - M
# calculate covariance matrix of centered matrix
V = cov(C.T)
# factorize covariance matrix
values, vectors = eig(V)
print(vectors)
print(values)
# project data
P = vectors.T.dot(C.T)
print(P.T)
```

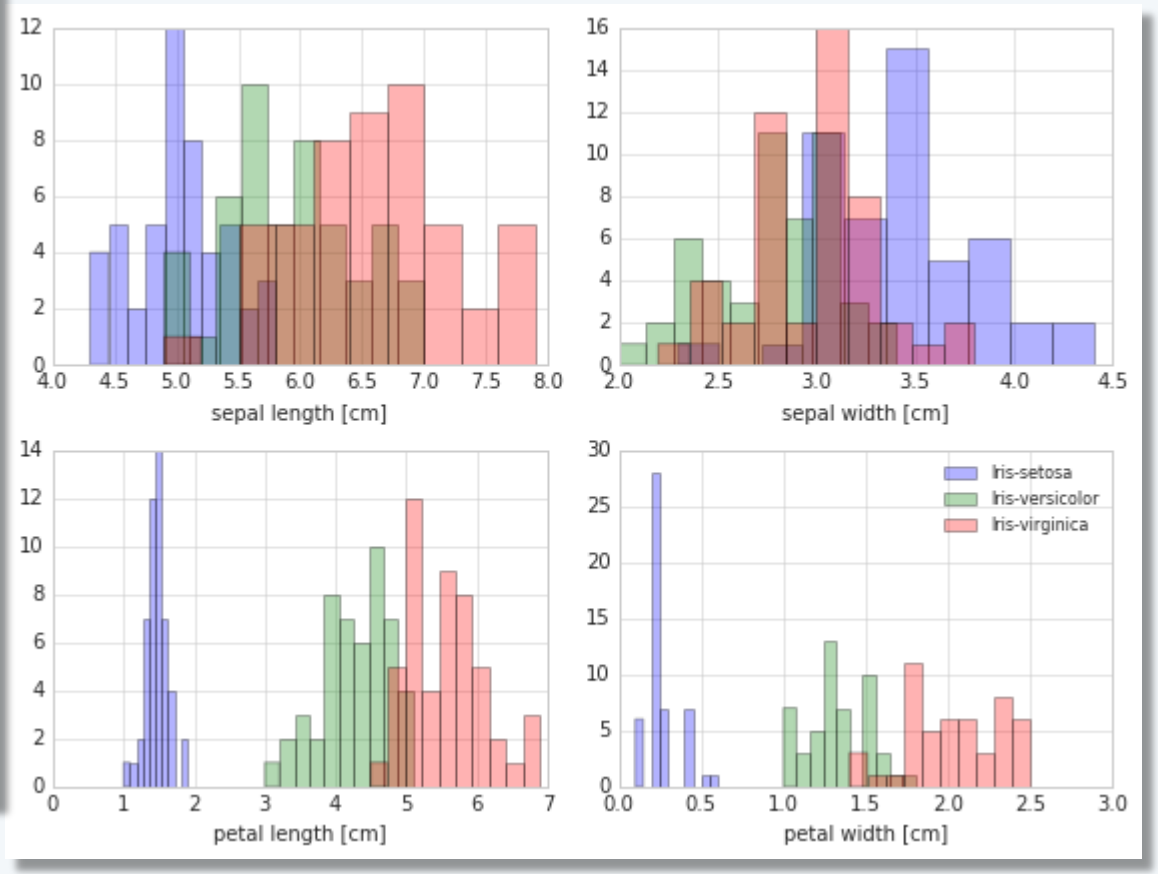
Iris Dataset - Exploratory Visualization (https://en.wikipedia.org/wiki/Iris_flower_data_set)

https://sebastianraschka.com/Articles/2015_pca_in_3_steps.html

```
from matplotlib import pyplot as plt
import numpy as np
import math

label_dict = {1: 'Iris-Setosa',
              2: 'Iris-Versicolor',
              3: 'Iris-Virgnica'}

feature_dict = {0: 'sepal length [cm]',
                1: 'sepal width [cm]',
                2: 'petal length [cm]',
                3: 'petal width [cm]'}

with plt.style.context('seaborn-whitegrid'):
    plt.figure(figsize=(8, 6))
    for cnt in range(4):
        plt.subplot(2, 2, cnt+1)
        for lab in ('Iris-setosa', 'Iris-versicolor', 'Iris-virginica'):
            plt.hist(X[y==lab, cnt],
                     label=lab,
                     bins=10,
                     alpha=0.3,)
        plt.xlabel(feature_dict[cnt])
    plt.legend(loc='upper right', fancybox=True, fontsize=8)

    plt.tight_layout()
    plt.show()
```

The three classes in the Iris dataset are:

Iris-setosa (n=50), blue
Iris-versicolor (n=50), green
Iris-virginica (n=50), red



Standardizing:

Since PCA yields a feature subspace that maximizes the variance along the axes, it makes (often) sense to standardize the data, especially, if it was measured on different scales. -> mean 0, variance 1

```
from sklearn.preprocessing import StandardScaler
X_std = StandardScaler().fit_transform(X)
```

Covariance Matrix:

```
import numpy as np
mean_vec = np.mean(X_std, axis=0)
cov_mat = (X_std - mean_vec).T.dot((X_std - mean_vec)) / (X_std.shape[0]-1)
print('Covariance matrix \n%s' %cov_mat)
```

```
Covariance matrix
[[ 1.00671141 -0.11010327  0.87760486  0.82344326]
 [-0.11010327  1.00671141 -0.42333835 -0.358937  ]
 [ 0.87760486 -0.42333835  1.00671141  0.96921855]
 [ 0.82344326 -0.358937    0.96921855  1.00671141]]
```

Diagonal not 1? -> Bessel

Eigenvectors (the Principal Components, see matrix P of the transformation)

```
cov_mat = np.cov(X_std.T)

eig_vals, eig_vecs = np.linalg.eig(cov_mat)

print('Eigenvectors \n%s' %eig_vecs)
print('\nEigenvalues \n%s' %eig_vals)
```

```
Eigenvectors
[[ 0.52237162 -0.37231836 -0.72101681  0.26199559]
 [-0.26335492 -0.92555649  0.24203288 -0.12413481]
 [ 0.58125401 -0.02109478  0.14089226 -0.80115427]
 [ 0.56561105 -0.06541577  0.6338014   0.52354627]]

Eigenvalues
[ 2.93035378  0.92740362  0.14834223  0.02074601]
```

Another note of caution:

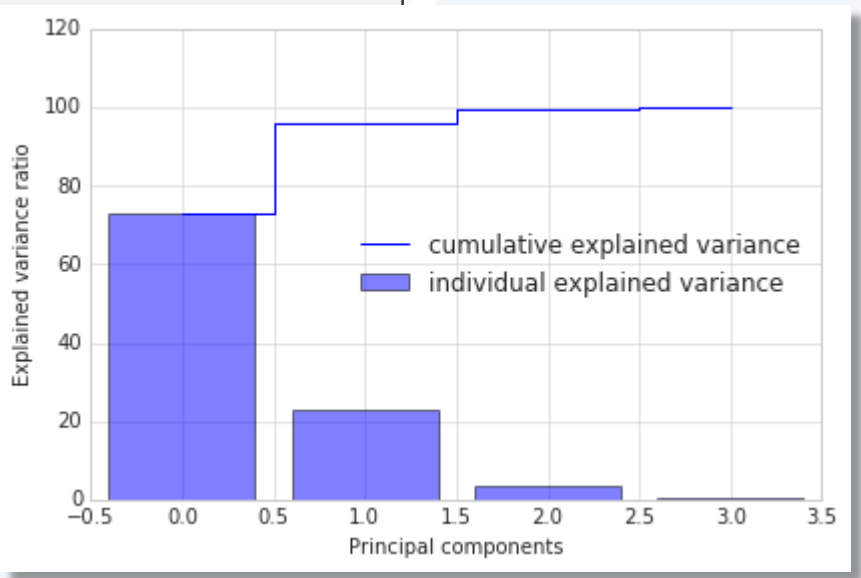
v : *(..., M, M) array*

The normalized (unit "length") eigenvectors, such that the column `v[:,i]` is the eigenvector corresponding to the eigenvalue `w[i]`.

Sorting Eigenvectors

```
# Make a list of (eigenvalue, eigenvector) tuples
eig_pairs = [(np.abs(eig_vals[i]), eig_vecs[:,i]) for i in range(len(eig_vals))]

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

# Visually confirm that the list is correctly sorted by decreasing eigenvalues
print('Eigenvalues in descending order:')
for i in eig_pairs:
    print(i[0])
```

```
Eigenvalues in descending order:
2.91081808375
0.921220930707
0.147353278305
0.0206077072356
```



Explained variance in dataset:The first two principal components contain 95.8% of the information.

Projection onto the new feature space

Here, we are reducing the 4-dimensional feature space to a 2-dimensional feature subspace, by choosing the “top 2” eigenvectors

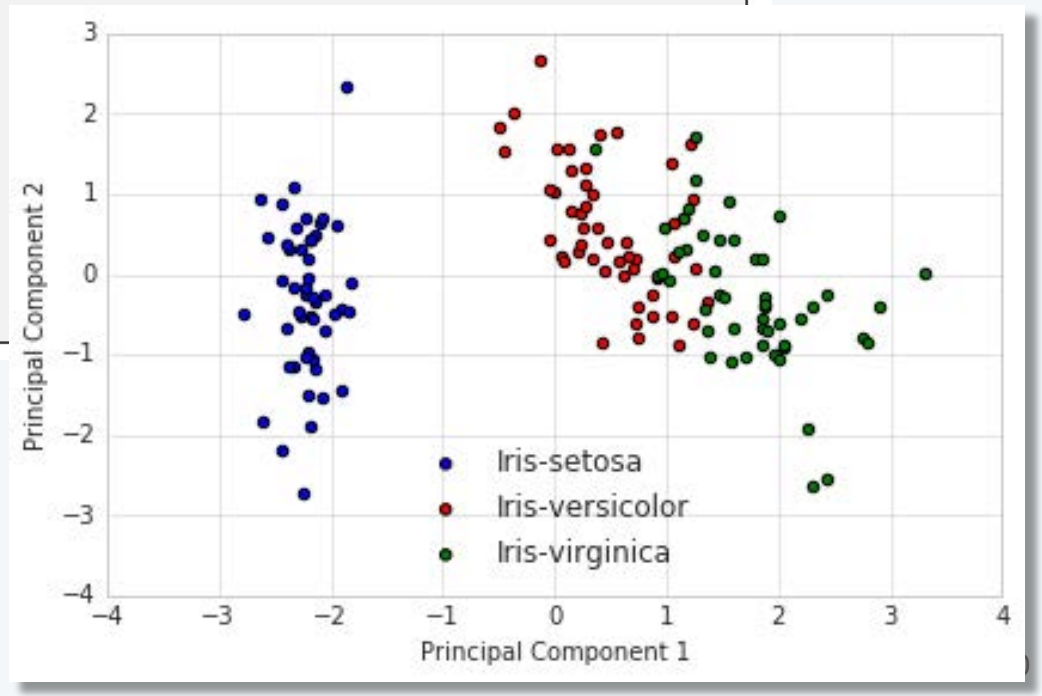
```
matrix_w = np.hstack((eig_pairs[0][1].reshape(4,1),
                      eig_pairs[1][1].reshape(4,1)))

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

```
Matrix W:
 [[ 0.52237162 -0.37231836]
 [-0.26335492 -0.92555649]
 [ 0.58125401 -0.02109478]
 [ 0.56561105 -0.06541577]]
```

Projection onto the new feature space:

```
Y = X_std.dot(matrix_w)
```

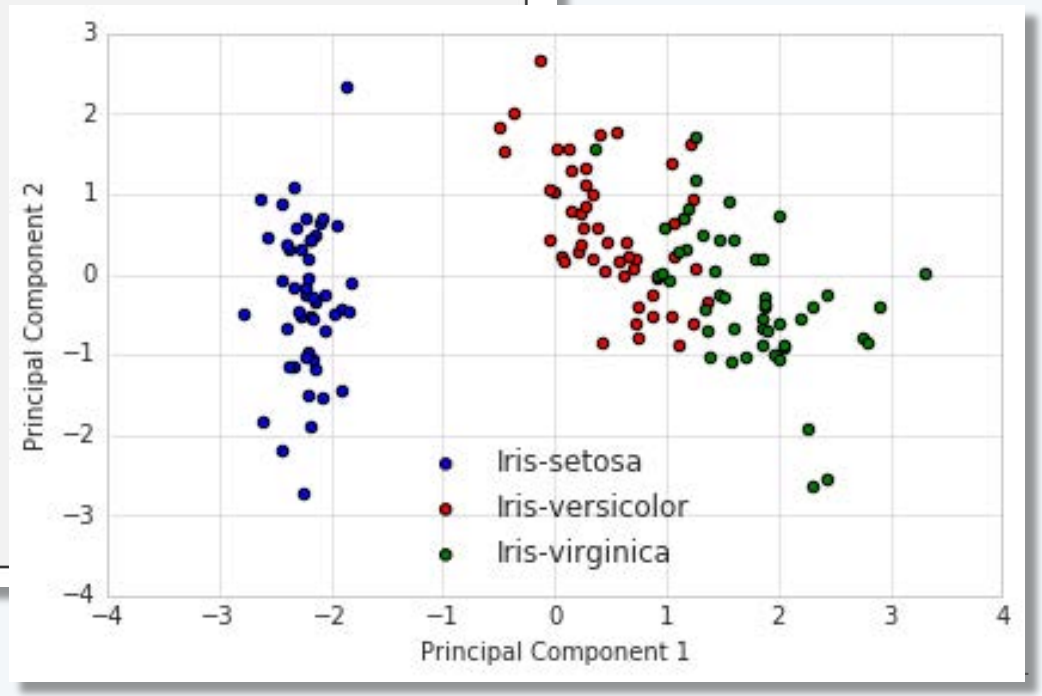
```
with plt.style.context('seaborn-whitegrid'):
    plt.figure(figsize=(6, 4))
    for lab, col in zip(('Iris-setosa', 'Iris-versicolor', 'Iris-virginica'),
                        ('blue', 'red', 'green')):
        plt.scatter(Y[y==lab, 0],
                    Y[y==lab, 1],
                    label=lab,
                    c=col)
    plt.xlabel('Principal Component 1')
    plt.ylabel('Principal Component 2')
    plt.legend(loc='lower center')
    plt.tight_layout()
    plt.show()
```

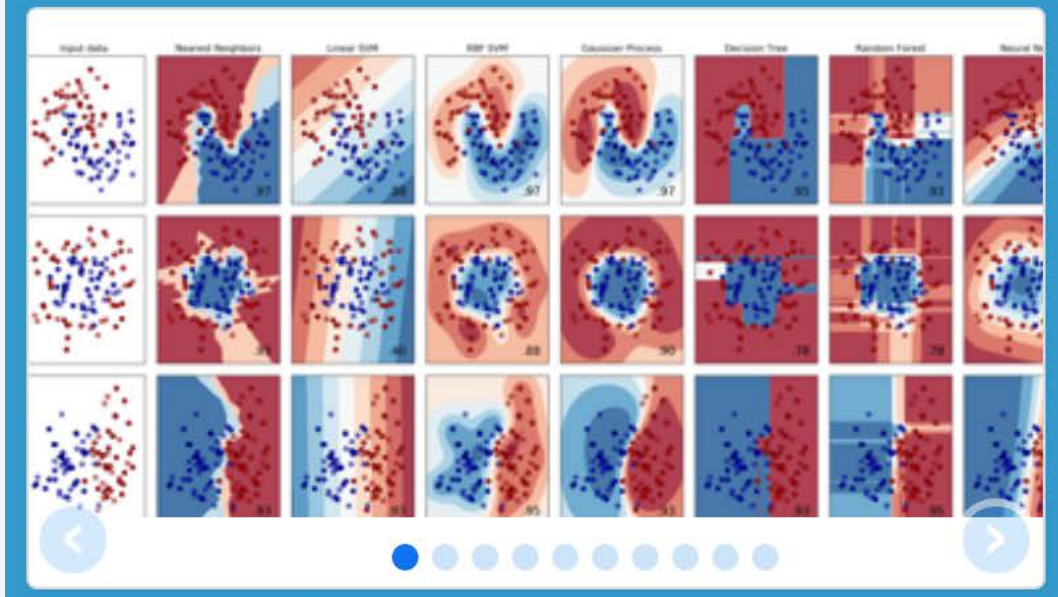


All in one go (here: **sklearn**)

```
from sklearn.decomposition import PCA as sklearnPCA
sklearn_pca = sklearnPCA(n_components=2)
Y_sklearn = sklearn_pca.fit_transform(X_std)
```

```
with plt.style.context('seaborn-whitegrid'):
    plt.figure(figsize=(6, 4))
    for lab, col in zip(('Iris-setosa', 'Iris-versicolor', 'Iris-virginica'),
                        ('blue', 'red', 'green')):
        plt.scatter(Y_sklearn[y==lab, 0],
                    Y_sklearn[y==lab, 1],
                    label=lab,
                    c=col)
    plt.xlabel('Principal Component 1')
    plt.ylabel('Principal Component 2')
    plt.legend(loc='lower center')
    plt.tight_layout()
    plt.show()
```





scikit-learn

Machine Learning in Python

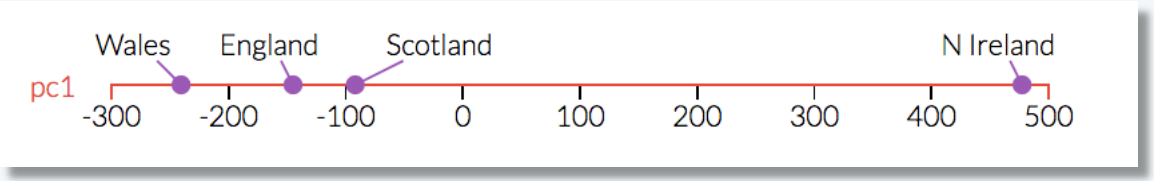
- Simple and efficient tools for data mining and data analysis
- Accessible to everybody, and reusable in various contexts
- Built on NumPy, SciPy, and matplotlib
- Open source, commercially usable - BSD license

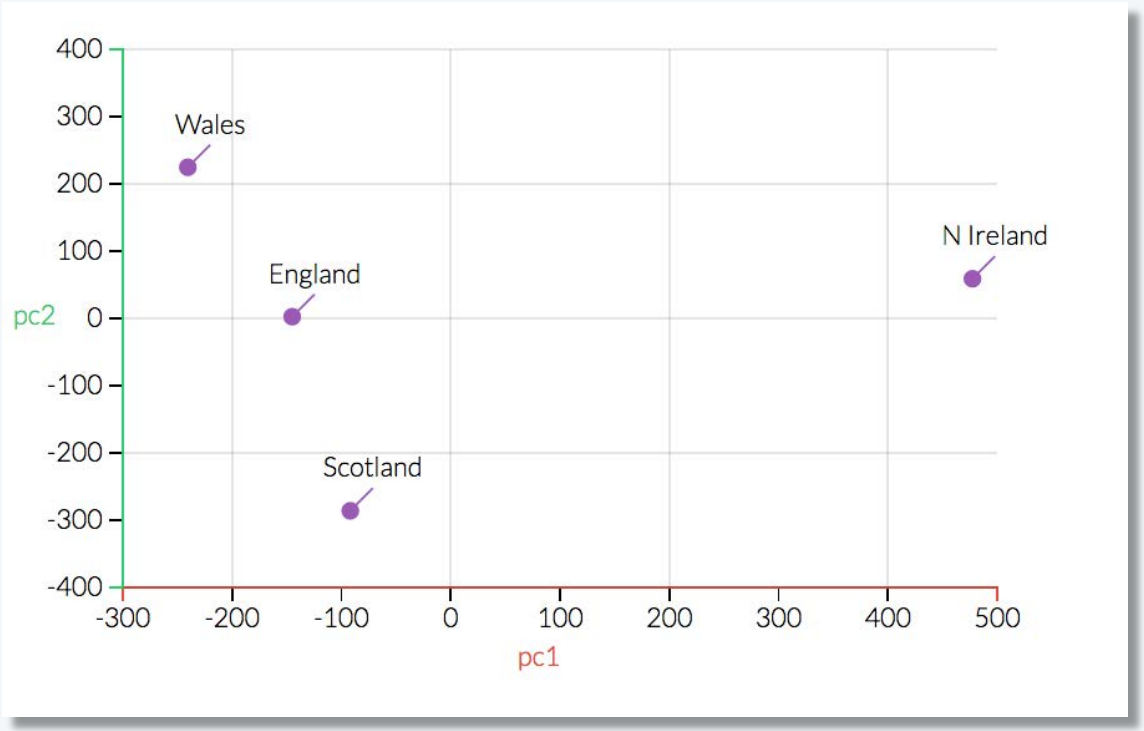
https://scikit-learn.org/

Eating in the UK (a 17d example)

	England	N Ireland	Scotland	Wales
Alcoholic drinks	375	135	458	475
Beverages	57	47	53	73
Carcase meat	245	267	242	227
Cereals	1472	1494	1462	1582
Cheese	105	66	103	103
Confectionery	54	41	62	64
Fats and oils	193	209	184	235
Fish	147	93	122	160
Fresh fruit	1102	674	957	1137
Fresh potatoes	720	1033	566	874
Fresh Veg	253	143	171	265
Other meat	685	586	750	803
Other Veg	488	355	418	570
Processed potatoes	198	187	220	203
Processed Veg	360	334	337	365
Soft drinks	1374	1506	1572	1256
Sugars	156	139	147	175

1d and 2d Score Plot:





Northern Irish eat way more grams of fresh potatoes and way fewer of fresh fruits, cheese, fish and alcoholic drinks.

PCA on pictures. Let's start with 64x64 dimensions.

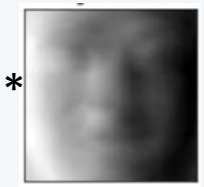
1. Obtain images
2. Represent every image as vector
3. Compute the average face
4. Subtract the average face
5. Compute the covariance matrix
6. Use a SVD decomposition / computation of eigenvectors
7. Project picture into eigenface-space and measure distance

We will use OpenCV (via Python bindings)

OpenCV (Open Source Computer Vision Library) is an open source computer vision and machine learning software library. https://opencv.org/

Eigenfaces (eigenvectors / principal components, visualized)


= 0.3845*

-0.5435*

+0.0345*

-0.74645*

+…

Using n=50 principal components

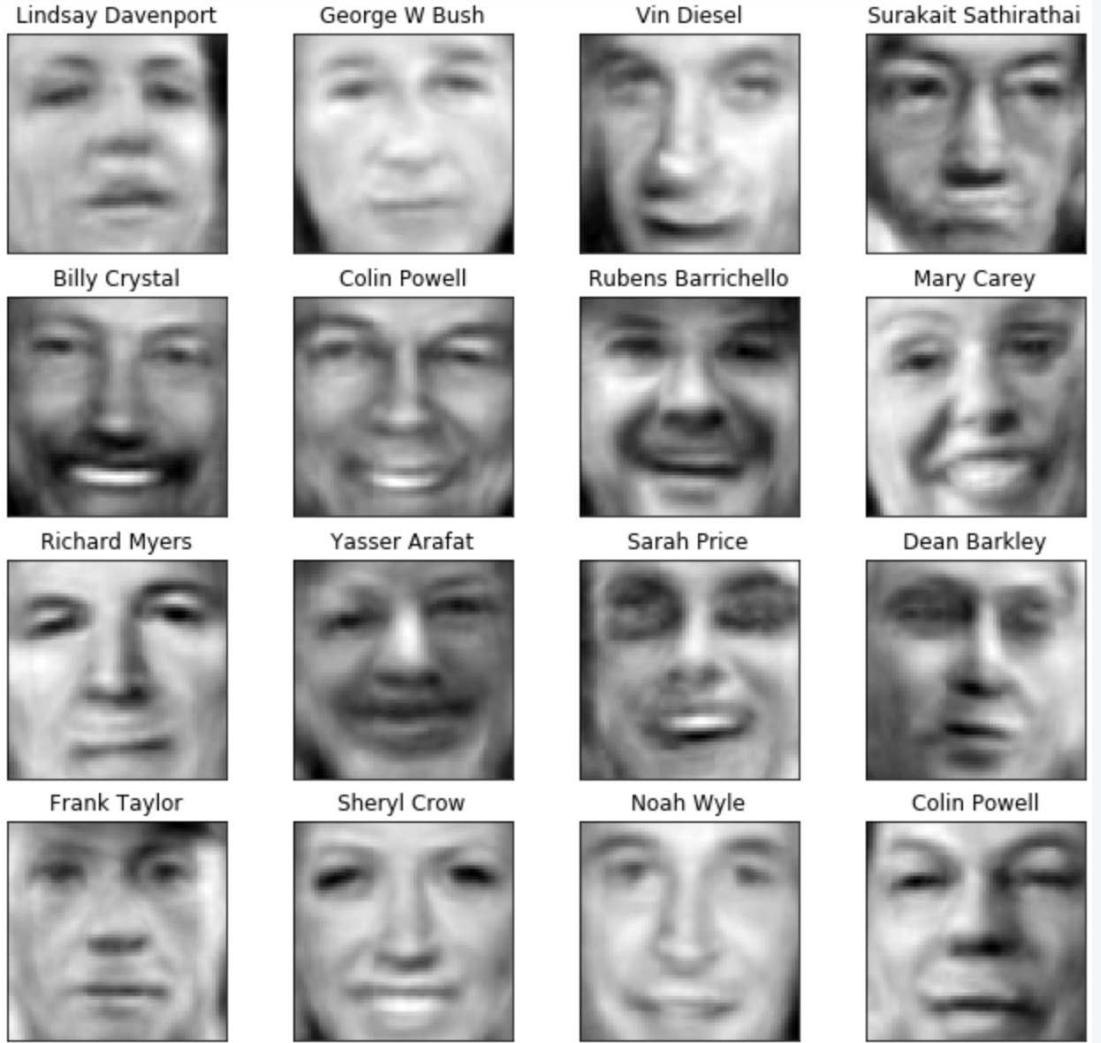
Original

Weighted combination of n eigenfaces

Using n=100 principal components

Original

Weighted combination of n eigenfaces

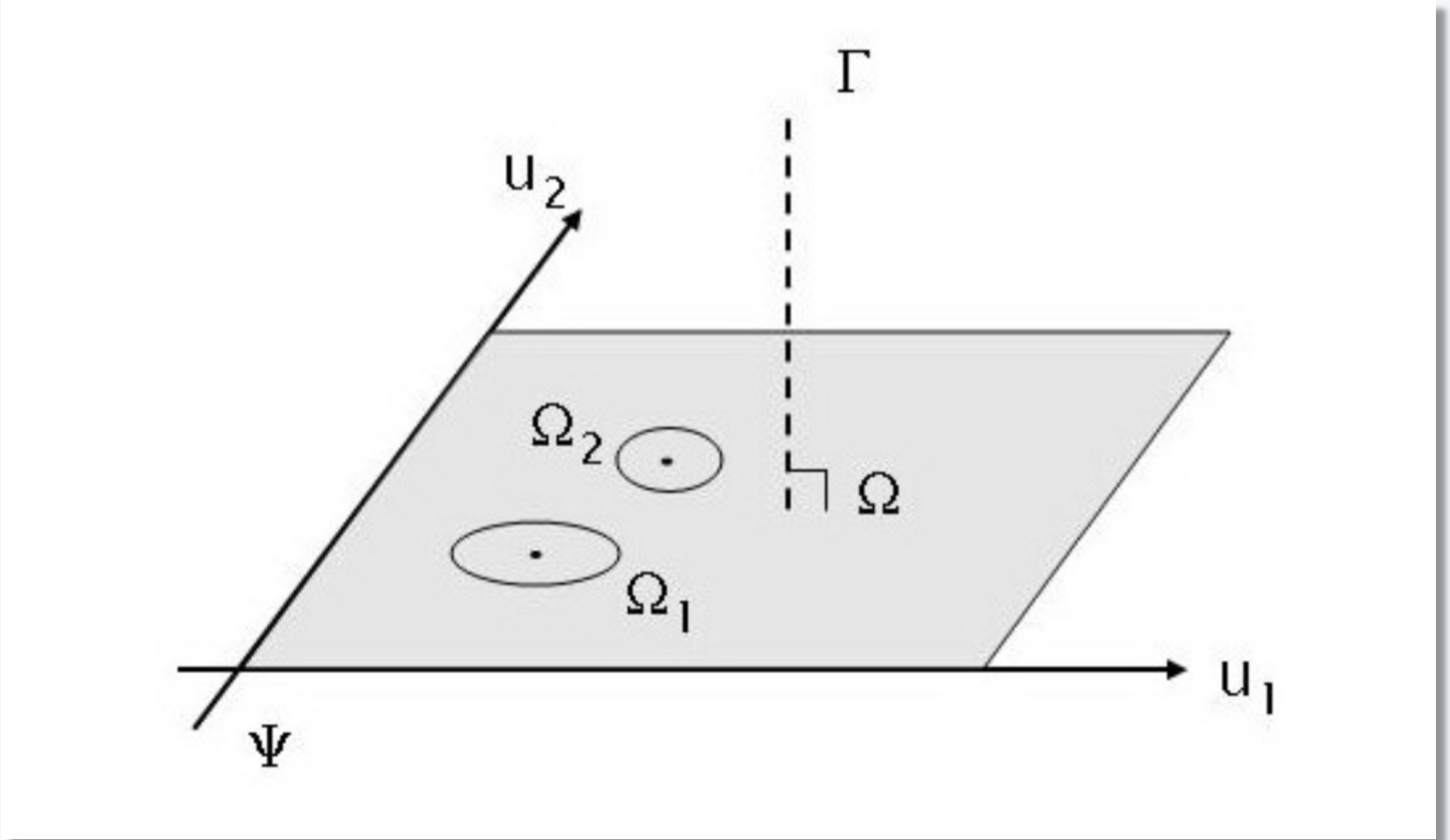


Using n=250 principal components

Original

Weighted combination of n eigenfaces

Eigenface space: origin is the average face, and axes are the eigenfaces.



Projection onto eigenface space.

…and using the distance to the eigenface space for classification

PCA::project

Projects vector(s) to the principal component subspace.

C++: Mat `PCA::project`(InputArray **vec**) `const` ¶

C++: void `PCA::project`(InputArray **vec**, OutputArray **result**) `const`

Python: `cv2.PCAProject`(data, mean, eigenvectors[, result]) → result

Parameters:
- **vec** – input vector(s); must have the same dimensionality and the same layout as the input data used at PCA phase, that is, if `CV_PCA_DATA_AS_ROW` are specified, then `vec.cols==data.cols` (vector dimensionality) and `vec.rows` is the number of vectors to project, and the same is true for the `CV_PCA_DATA_AS_COL` case.
- **result** – output vectors; in case of `CV_PCA_DATA_AS_COL`, the output matrix has as many columns as the number of input vectors, this means that `result.cols==vec.cols` and the number of rows match the number of principal components (for example, `maxComponents` parameter passed to the constructor).

The methods project one or more vectors to the principal component subspace, where each vector projection is represented by coefficients in the principal component basis. The first form of the method returns the matrix that the second form writes to the result. So the first form can be used as a part of expression while the second form can be more efficient in a processing loop.

PCA::backProject

Reconstructs vectors from their PC projections.

C++: Mat `PCA::backProject`(InputArray **vec**) `const`

C++: void `PCA::backProject`(InputArray **vec**, OutputArray **result**) `const`

Python: `cv2.PCABackProject`(data, mean, eigenvectors[, result]) → result

Parameters:
- **vec** – coordinates of the vectors in the principal component subspace, the layout and size are the same as of `PCA::project` output vectors.
- **result** – reconstructed vectors; the layout and size are the same as of `PCA::project` input vectors.

The methods are inverse operations to `PCA::project()`. They take PC coordinates of projected vectors and reconstruct the original vectors. Unless all the principal components have been retained, the reconstructed vectors are different from the originals. But typically, the difference is small if the number of components is large enough (but still much smaller than the original vector dimensionality). As a result, PCA is used.

Example, using a 10 dimensional eigenface space:

```
In [16]: i=(data[0,:])

In [17]: result=cv2.PCAProject(i,mean.reshape(i.shape),eigenVectors)

In [18]: i.shape
Out[18]: (96768,)

In [19]: result
Out[19]:
array([[ 36.499718],
       [ 24.19416 ],
       [-16.804926],
       [-31.866632],
       [-25.335268],
       [ -7.799864],
       [ -5.968835],
       [ 11.209648],
       [-22.533894],
       [-16.296684]], dtype=float32)
```

Note: each picture is represented with 10, instead of 96768 numbers (but of course there is quality loss)

Back projection:

```
In [21]: pic=cv2.PCABackProject(result, mean.reshape(i.shape), eigenVectors).reshape(sz)

In [22]: pic.shape
Out[22]: (192, 168, 3)
```

Online Demonstration

based on https://www.learnopencv.com/eigenface-using-opencv-c-python/

```
Pic:1 Distance: 28.477287449985468
Pic:2 Distance: 19.323155501490803
Pic:3 Distance: 28.21498025669547
Pic:4 Distance: 34.18605562687379
Pic:5 Distance: 34.562687827146206
Pic:6 Distance: 27.64321237071724
Pic:7 Distance: 27.55846148653267
Pic:8 Distance: 25.07270763191366
Pic:9 Distance: 26.27102349445904
Pic:10 Distance: 21.028263689708268
Pic:11 Distance: 66.85554003501879
Pic:12 Distance: 37.01324926579617
Pic:13 Distance: 19.793292976240558
Pic:14 Distance: 32.157923569195624
Pic:15 Distance: 29.32366880478148
Pic:16 Distance: 15.332240101064556
Pic:17 Distance: 30.530705542314877
Pic:18 Distance: 25.80040534357497
Pic:19 Distance: 32.19750910756968
```

Here: Using the square root of the sum of squared absolute distances (10-dimensional eigenface space).

Which was the dog?

Online Demonstration

```
In [3]: mean, eigenVectors, eigenvalues = cv2.PCACompute2(data[10:], mean=None, maxComponents=100)
```

```
Pic:2 Distance: 14.036487910761648
Pic:3 Distance: 17.77061960293203
Pic:4 Distance: 16.387858749367638
Pic:5 Distance: 12.455587164776626
Pic:6 Distance: 15.34815783577464
Pic:7 Distance: 15.307848737038025
Pic:8 Distance: 20.074247584446205
Pic:9 Distance: 7.665959987673816
Pic:10 Distance: 10.806869633746027
Pic:11 Distance: 50.60634700494516
Pic:12 Distance: 12.313413199558708
Pic:13 Distance: 11.968826988035847
Pic:14 Distance: 11.669270298838166
Pic:15 Distance: 12.179563882892865
Pic:16 Distance: 10.627303393177312
Pic:17 Distance: 12.078693666525941
Pic:18 Distance: 11.506397512158754
Pic:19 Distance: 12.253399758379327
```

Here: 100-dimensional eigenface space.

Which was the dog?

Conclusion

- PCA: a **very important statistical procedure** (some might say machine learning method) for dimensionality reduction.
- Revealing the **internal structure of the data** in a way that best explains the variance in the data.
- Used in (as such or as a basis for) a **whole range of application scenarios**, such as stock market predictions, the analysis of gene expression data, neuroscience, chemical compound analysis, and many many more.

Visual Intro: http://setosa.io/ev/principal-component-analysis/
A Pyhton based PCA tutorial (sklearn): https://sebastianraschka.com/Articles/2015_pca_in_3_steps.html
Eigenfaces with OpenCV: https://www.learnopencv.com/eigenface-using-opencv-c-python/