08_LinearAlgebra

January 21, 2025

1 Lab 08: Linear Algebra

Table of contents:

1.1 1. Algorithmic Complexity (time and space complexity)

1.1.1 Time complexity

Profiling (e.g. with `timeit`) doesn't tell us much about how an algorithm will perform on a different computer since it is determined by the hardware features. To compare performance in a device-indpendent fashion, a formalism (a.k.a the "Big-O") is used that characterizes functions in terms of their rates of growth as a function of the size n of the input.

An algorithm is compared to a given function $g(n)$ with a well defined scaling with n, e.g. n^2; if the ratio of the two is bounded, than that algorithm is $O(g(n))$. Note that: * Only the largest terms in the scaling of $g(n)$ is kept in the notation * two algorithms can have the same complexity and have very different performance; the same complexity only implies that the difference in performance is independent of n.

Comparing bubble sort $O(n^2)$ and merge sort $O(n \log n)$

[2]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

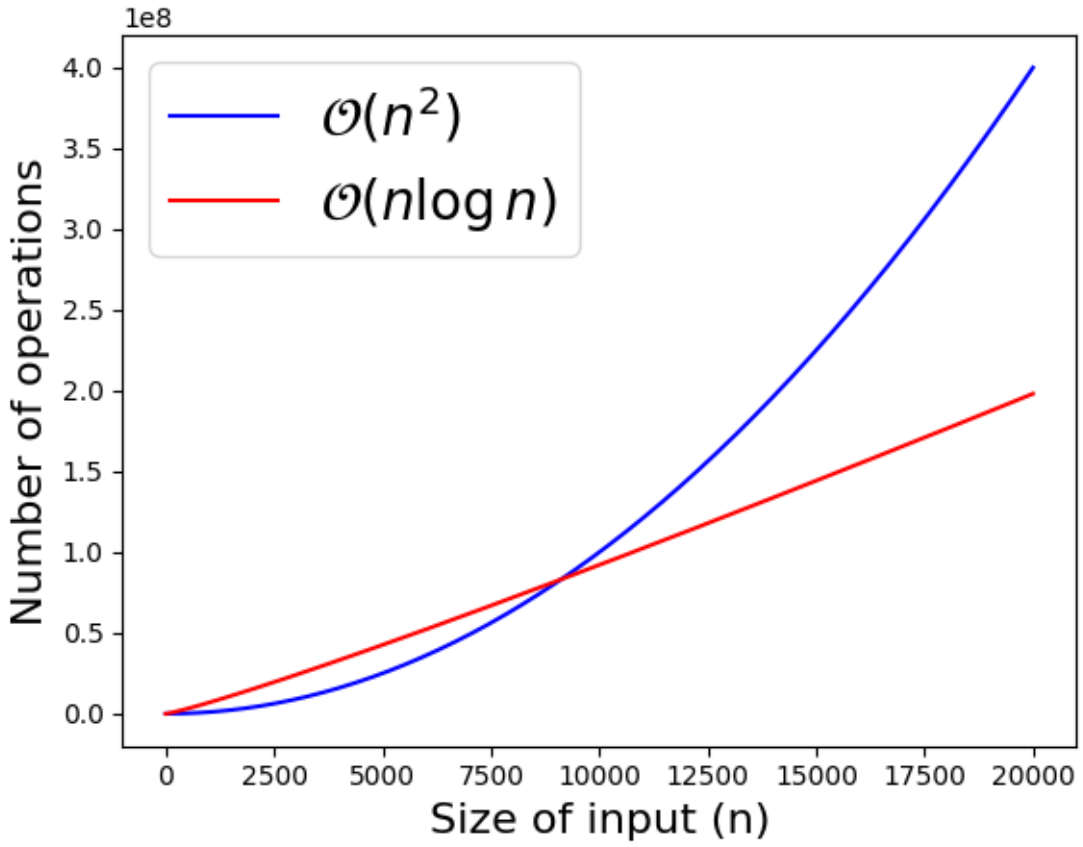
[2]:
```
def f1(n, k):
    return k*n*n

def f2(n, k):
    return k*n*np.log(n)

n = np.arange(0.1, 20001)
```

```
plt.plot(n, f1(n, 1), c='blue')
plt.plot(n, f2(n, 1000), c='red')
plt.xlabel('Size of input (n)', fontsize=16)
plt.ylabel('Number of operations', fontsize=16)
plt.legend(['$\mathcal{O}(n^2)$', '$\mathcal{O}(n \log n)$'], loc='best',␣
↪fontsize=20);
```

```
<>:12: SyntaxWarning: invalid escape sequence '\m'
<>:12: SyntaxWarning: invalid escape sequence '\m'
<>:12: SyntaxWarning: invalid escape sequence '\m'
<>:12: SyntaxWarning: invalid escape sequence '\m'
/var/folders/vk/kftm8379123bsmwrdp8l0xr00000gn/T/ipykernel_3981/3771379069.py:12
: SyntaxWarning: invalid escape sequence '\m'
  plt.legend(['$\mathcal{O}(n^2)$', '$\mathcal{O}(n \log n)$'], loc='best',
fontsize=20);
/var/folders/vk/kftm8379123bsmwrdp8l0xr00000gn/T/ipykernel_3981/3771379069.py:12
: SyntaxWarning: invalid escape sequence '\m'
  plt.legend(['$\mathcal{O}(n^2)$', '$\mathcal{O}(n \log n)$'], loc='best',
fontsize=20);
```



See here for the complexity of operations on standard Python data structures. Note for instance

that searching a list is much more expensive than searching a dictionary.

Here a few examples:

O(1) - Constant Time Complexity An algorithm with constant time complexity performs the same number of steps regardless of input size.

[ ]:
```
def get_first_element(array):
    return array[0]  # Always takes one step.
```

O(n) - Linear Time Complexity An algorithm with linear time complexity performs a number of steps proportional to the size of the input.

[ ]:
```
def print_pairs(array):
    for i in range(len(array)):
        for j in range(len(array)):
            print(array[i], array[j])  # Prints every pair of elements.
```

O(n^2) - Quadratic Time Complexity An algorithm with quadratic time complexity often involves nested loops.

[ ]:
```
def print_pairs(array):
    for i in range(len(array)):
        for j in range(len(array)):
            print(array[i], array[j])  # Prints every pair of elements.
```

O(log n) - Logarithmic Time Complexity Logarithmic complexity arises when the problem size reduces by a constant factor in each step, such as binary search.

[ ]:
```
def binary_search(array, target):
    left, right = 0, len(array) - 1
    while left <= right:
        mid = (left + right) // 2
        if array[mid] == target:
            return mid  # Found the target.
        elif array[mid] < target:
            left = mid + 1  # Search in the right half.
        else:
            right = mid - 1  # Search in the left half.
    return -1  # Target not found.
```

O(n log n) - Linearithmic Time Complexity Sorting algorithms like merge sort and quicksort have $O(n \log n)$ complexity.

[ ]:
```
def merge_sort(array):
    if len(array) <= 1:
        return array
    mid = len(array) // 2
```

```
    left = merge_sort(array[:mid])
    right = merge_sort(array[mid:])
    return merge(left, right)
```

[ ]:
```
def merge(left, right):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result
```

O(2^n) - Exponential Time Complexity An algorithm with exponential complexity often explores all possible combinations, such as solving the Traveling Salesman Problem with brute force, where 2^n arises because for n cities, with the algorithm essentially exploring all subsets of cities as part of the recursion.

[ ]:
```
def tsp_brute_force(graph, current, visited, start):
    if len(visited) == len(graph):  # If all cities are visited
        return graph[current][start]  # Return to the start city

    min_cost = float('inf')
    for next_city in graph:
        if next_city not in visited:  # Visit unvisited cities
            visited.add(next_city)
            cost = graph[current][next_city] + tsp_brute_force(graph,␣
 ↪next_city, visited, start)
            min_cost = min(min_cost, cost)
            visited.remove(next_city)
    return min_cost
```

O(n!) - Factorial Time Complexity The Traveling Salesman Problem can be implemented also witha factorial complexity if it explicitly involves computing the cost for every permutation of the cities, which are $n!$

[ ]:
```
from itertools import permutations

def traveling_salesman(graph, start):
    nodes = list(graph.keys())
    nodes.remove(start)
    shortest_path = float('inf')
```

```
    for perm in permutations(nodes):
        current_path_weight = 0
        current_node = start
        for next_node in perm:
            current_path_weight += graph[current_node][next_node]
            current_node = next_node
        current_path_weight += graph[current_node][start]  # Return to start
        shortest_path = min(shortest_path, current_path_weight)
    return shortest_path
```

1.1.2 Space Complexity

We can also use O notation in the same way to measure the space complexity of an algorithm. The notion of space complexity becomes important when your data volume is of the same magnitude or larger than the memory you have available. **In that case, an algorihtm with high space complexity may end up having to swap memory constantly, and will perform far worse than its time complexity would suggest.**

Just as you should have a good idea of how your algorithm will scale with increasing n, you should also be able to know how much memory your data structures will require. For example, if you had an $n \times p$ matrix of integers, an $n \times p$ matrix of floats, and an $n \times p$ matrix of complex floats, how large can n and p be before you run out of RAM to store them?

[ ]:
```
# Notice how much overhead Python objects have
# A raw integer should be 64 bits or 8 bytes only

import sys
print (sys.getsizeof(1))
print (sys.getsizeof(12345678901234567890123456789012345678901234567890))
print (sys.getsizeof(3.14))
print (sys.getsizeof(3j))
print (sys.getsizeof('a'))
print (sys.getsizeof('hello world'))
```

[ ]:
```
print (np.ones((100,100), dtype='byte').nbytes)
print (np.ones((100,100), dtype='i2').nbytes)
print (np.ones((100,100), dtype='int').nbytes) # default is 64 bits or 8 bytes
print (np.ones((100,100), dtype='f4').nbytes)
print (np.ones((100,100), dtype='float').nbytes) # default is 64 bits or 8 bytes
print (np.ones((100,100), dtype='complex').nbytes)
```

1.2 2. Linear algebra with scipy and numpy

Scipy: high-level scientific computing The `scipy` package contains various toolboxes dedicated to common issues in scientific computing. Its different submodules correspond to different applications, such as interpolation, integration, optimization, image processing, statistics, special functions, etc.

`scipy` can be compared to other standard scientific-computing libraries, such as the GSL (GNU

Scientific Library for C and C++), or Matlab's toolboxes. `scipy` is the core package for scientific routines in Python; it is meant to operate efficiently on numpy arrays, so that numpy and `scipy` work hand in hand.

Before implementing a routine, it is worth checking if the desired data processing is not already implemented in `scipy`. *As non-professional programmers, scientists often tend to re-invent the wheel, which leads to buggy, non-optimal, difficult-to-share and unmaintainable code. By contrast,* ***`scipy`****'s routines are optimized and tested, and should therefore be used when possible.*

Comments from class The `scipy.linalg` module provides standard linear algebra operations, relying on an underlying efficient implementation (BLAS, LAPACK). We will review a few examples and applications. Sometimes numpy implements those methods too: if a given algorithm is present both in numpy and **scipy**, **typically** the latter **is more performant**.

[3]:
```
import numpy as np
from scipy import linalg as la
from matplotlib import pyplot as plt

# to limit the printout
%precision 4
np.set_printoptions(suppress=True)
```

[ ]:
```
# norm of a vector
v = np.array([1,2])
print (la.norm(v))

# distance between two vectors
w = np.array([1,1])
print (la.norm(v-w))

# inner products
print (v.dot(w))
```

Elaborate example: covariance matrix as outer product The **inner product** is just matrix multiplication of a 1×n vector with an n×1 vector. The **outer product** of two vectors is instead just the opposite. It is given by:

$$v \otimes w = vw^t$$

Note that v and w are column vectors. The result of the inner product is a scalar. The result of the outer product is a matrix.

[ ]:
```
print (np.outer(v,w))
```

Suppose now we measure 4 quantities, each 10 times. Let's assume each quantity is flat distributed in the $[0-1]$ interval

[3]:
```
n, p = 10, 4
v = np.random.random((p,n))
v
```

[3]:
```
array([[0.37289867, 0.2610513 , 0.57839415, 0.39332893, 0.30273784,
        0.93058689, 0.02607439, 0.92200913, 0.33258836, 0.71552857],
       [0.64634393, 0.30102521, 0.76540247, 0.85786252, 0.86579385,
        0.76311493, 0.34987059, 0.33372394, 0.84699286, 0.48397251],
       [0.59205856, 0.35403518, 0.45624674, 0.01835578, 0.70092546,
        0.05036957, 0.80418008, 0.14080124, 0.99501448, 0.16188684],
       [0.41857323, 0.2126468 , 0.54372619, 0.97911428, 0.38662432,
        0.4839443 , 0.71164311, 0.54977293, 0.71083951, 0.15056849]])
```

Let's recall the definition of the covariance matrix:

$$\mathrm{Cov}(X_i, X_j) = \frac{\sum_{h=1}^{n}(X_{hi} - \bar{X}_i)(X_{hj} - \bar{X}_j)}{n-1}$$

with $\mathrm{Cov}(X, X)$ the variance of the variable X.

[ ]:
```
# Numpy has a built-in function to compute the covariance:
np.cov(v)
```

[ ]:
```
array([[ 0.08772816,  0.00135078, -0.06950777, -0.01563268],
       [ 0.00135078,  0.05399879,  0.01034869,  0.0226037 ],
       [-0.06950777,  0.01034869,  0.11493309,  0.00732588],
       [-0.01563268,  0.0226037 ,  0.00732588,  0.06060312]])
```

[7]:
```
# Let's check that this works as expected...

# Calculate the covariance explicilty:

# compute the mean of each sequence and set the right shape
v_mean= v.mean(1)[:, np.newaxis]
print (v_mean, end= '\n\n')

# re-center each sequence around its mean
w = v - v_mean
print (w, end= '\n\n')

# compute the covariance matrix
cov=w.dot(w.T)/(n - 1)

# They are the same
print (cov)
```

```
[[0.48351982]
 [0.62141028]
 [0.42738739]
```

```
 [0.51474532]]

[[-0.11062116 -0.22246852  0.09487433 -0.0901909  -0.18078198  0.44706707
  -0.45744543  0.4384893  -0.15093146  0.23200874]
 [ 0.02493365 -0.32038507  0.14399219  0.23645224  0.24438357  0.14170465
  -0.27153969 -0.28768634  0.22558258 -0.13743777]
 [ 0.16467117 -0.07335221  0.02885934 -0.40903161  0.27353807 -0.37701783
   0.37679269 -0.28658615  0.56762708 -0.26550055]
 [-0.09617209 -0.30209851  0.02898088  0.46436897 -0.128121   -0.03080101
   0.19689779  0.03502762  0.19609419 -0.36417683]]

[[ 0.08772816  0.00135078 -0.06950777 -0.01563268]
 [ 0.00135078  0.05399879  0.01034869  0.0226037 ]
 [-0.06950777  0.01034869  0.11493309  0.00732588]
 [-0.01563268  0.0226037   0.00732588  0.06060312]]
```

1.2.1 Traces and determinants

[ ]:
```
n = 6
M = np.random.randint(100,size=(n,n))
print(M,'\n')
print ('determinant:',la.det(M),'\n')
print ('trace:',M.trace(),'\n')
```

1.3 3. Matrix Decomposition

Often data analysis problems boil down to solving linear systems. An example is the Netflix Competition, where a matrix of 400000×18000 (ratings times movies) needed to be dealt with.

Matrix decompositions are an important step in solving linear systems in a computationally efficient manner.

1.3.1 Lower-Upper factorization

Let A be a square matrix. An LU factorization refers to the factorization of A, with proper row and/or column orderings or permutations, into two factors – a lower triangular matrix L and an upper triangular matrix U:

$A = LU$

when solving a system of linear equations, $Ax = b = LUx$, the solution is done in two logical steps: 1. solve $Ly = b$ for y. 2. solve $Ux = y$ for x.

Often a permutation P is needed (*partial pivoting*) to best reorder the rows of the original matrix

[ ]:
```
A = np.array([[1,3,4],[2,1,3],[4,1,2]])
print(A,'\n')

P, L, U = la.lu(A)
print(np.dot(P.T, A),'\n')
```

```
print(np.dot(L, U),'\n')
print(P,'\n')
print(L,'\n')
print(U,'\n')
```

1.3.2 Eigendecomposition

Given an $n \times n$ matrix A, with $\det A \neq 0$, then there exist n linearly independent eigenvectors and A may be decomposed in the following manner:

$$A = V\Lambda V^{-1}$$

where Λ is a diagonal matrix whose diagonal entries are the eigenvalues of A and the columns of V are the corresponding eigenvectors of A.

Eigenvalues are roots of the *characteristic polynomial* of A:

$$\det(A - \lambda I) = 0$$

[6]:
```
A = np.array([[0,1,1],[2,1,0],[3,4,5]])
print (A)
print(la.det(A),'\n')


l, V = la.eig(A)
print (V, end = '\n\n')
print (l, end = '\n\n')
print(np.real_if_close(l))
```

```
[[0 1 1]
 [2 1 0]
 [3 4 5]]
-5.0

[[ 0.1802  0.6721 -0.    ]
 [ 0.0743 -0.7249 -0.7071]
 [ 0.9808  0.1509  0.7071]]

[ 5.8541+0.j -0.8541+0.j  1.    +0.j]

[ 5.8541 -0.8541  1.    ]
```

[ ]:
```
print(np.dot(V,np.dot(np.diag(np.real_if_close(l)), la.inv(V))),'\n')
```

1.3.3 Singular Value Decomposition (<- new thing! study!)

Way to deal with non-square matrices and kind of "diagonalize" them

Another important matrix decomposition is singular value decomposition or SVD. For any $m \times n$ matrix A, we may write:

$$A = UDV^T$$

where U is a orthogonal $m \times m$ matrix, D (spectrum) is a rectangular, diagonal $m \times n$ matrix with diagonal entries $d_1, \ldots, d_m$ all non-negative, V is an orthogonal $n \times n$ matrix.

in data analysis, m = rows = number of data points (huge), n = cols = number of features (usually not more than a few hundreds)

The singular-value decomposition is a generalization of the eigendecomposition in the sense that it can be applied to any $m \times n$ matrix whereas eigenvalue decomposition can only be applied to diagonalizable matrices.

Given an SVD of A, as described above, the following holds:

$$A^T A = VD^T U^T UDV^T = VD^T DV^T$$
$$AA^T = UD^T V^T VDU^T = UD^T DU^T$$

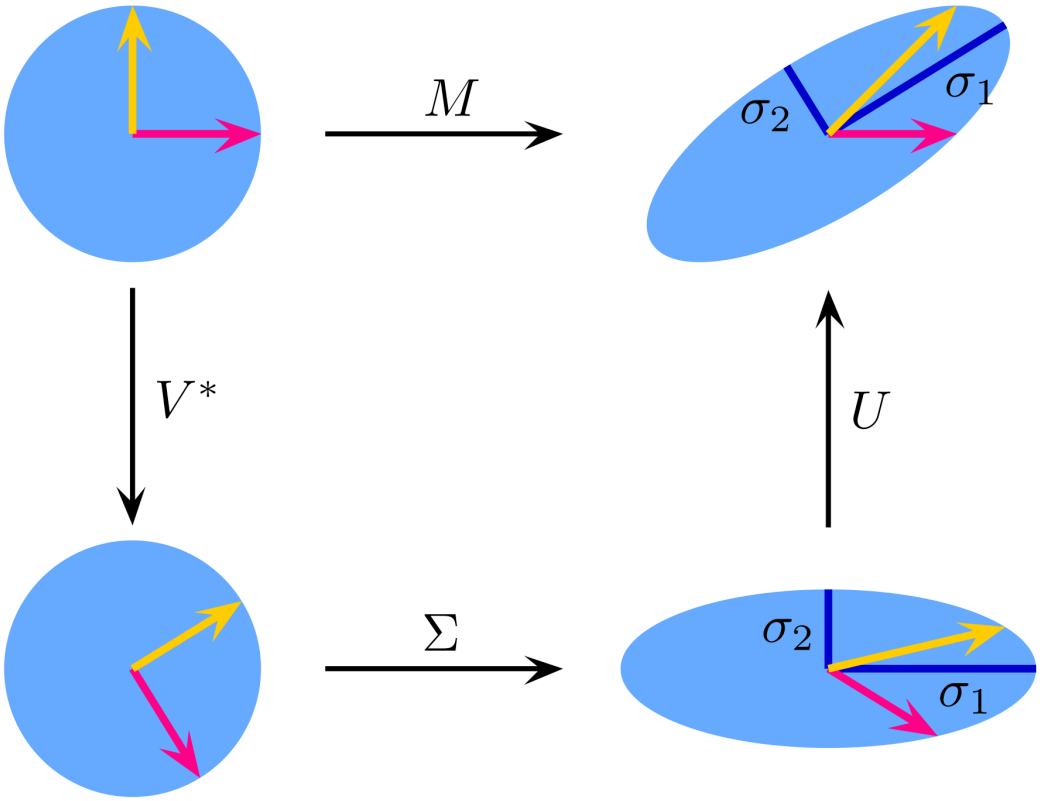
The right-hand sides of these relations describe the eigenvalue decompositions of the left-hand sides. Consequently: * The columns of V (right-singular vectors) are eigenvectors of $A^T A$. * The columns of U (left-singular vectors) are eigenvectors of AA^T. * The non-zero elements of D (non-zero singular values) are the square roots of the non-zero eigenvalues of $A^T A$ or AA^T.

A geometrical representation of SVD is given by the following figure:

CHECK AGAIN!

[1]:
```
from IPython.display import Image
Image("Singular-Value-Decomposition.png", width = 300, height= 300)
```

[1]:



$$M = U \cdot \Sigma \cdot V^*$$

[ ]:
```
m, n = 5, 4
A = np.random.randn(m, n) #+ 1.j*np.random.randn(m, n)
print (A,'\n')

U, spectrum, Vt = la.svd(A)

print("shapes:", U.shape,  spectrum.shape, Vt.shape)

print (spectrum,'\n')
print (U,'\n')
print (Vt,'\n')
```

[ ]:
```
# Let's verify the definition of SVD by hand
D = np.zeros((m, n))
for i in range(min(m, n)):
```

```
    D[i, i] = spectrum[i]
SVD = np.dot(U, np.dot(D, Vt))
print (SVD)
np.allclose(SVD, A)
```

Clearly scipy provide already a "solve" method for the linear systems of the type:

$$Ax = b$$

still, knowing a little bit what are the algorithms underneath comes handy sometimes, e.g. the solve method can be instructed about what kind of matrix A is likely to be (symmetric, hermitian, positive definite, etc.)

[ ]:
```
A = np.array([[3, 2, 0], [1, -1, 0], [0, 5, 1]])
b = np.array([2, 4, -1])
x = la.solve(A, b)
print (x)
```

[ ]:
```
np.dot(A, x) == b
```

1.4 3. Principal Component Analysis

Principal Components Analysis (PCA) aims at **finding** and **ranking** all the **eigenvalues and eigenvectors** of a given **dataset's covariance matrix**.

This is **useful because** high-dimensional data (with p features) **may have nearly all their variation in a small number of dimensions** k, i.e. in the **subspace** spanned by the eigenvectors of the covariance matrix that have the k **largest eigenvalues**.

a feature with low variance, or a feature that is uncorrelated with the others, is typically irrelevant in your data analysis

If we project the original data into this subspace, we can have a **dimension reduction** (from p to k) **with** (hopefully) **little loss of information**.

Numerically, PCA can be done - either by means of eigendecomposition on the covariance matrix - or via SVD on the data matrix.

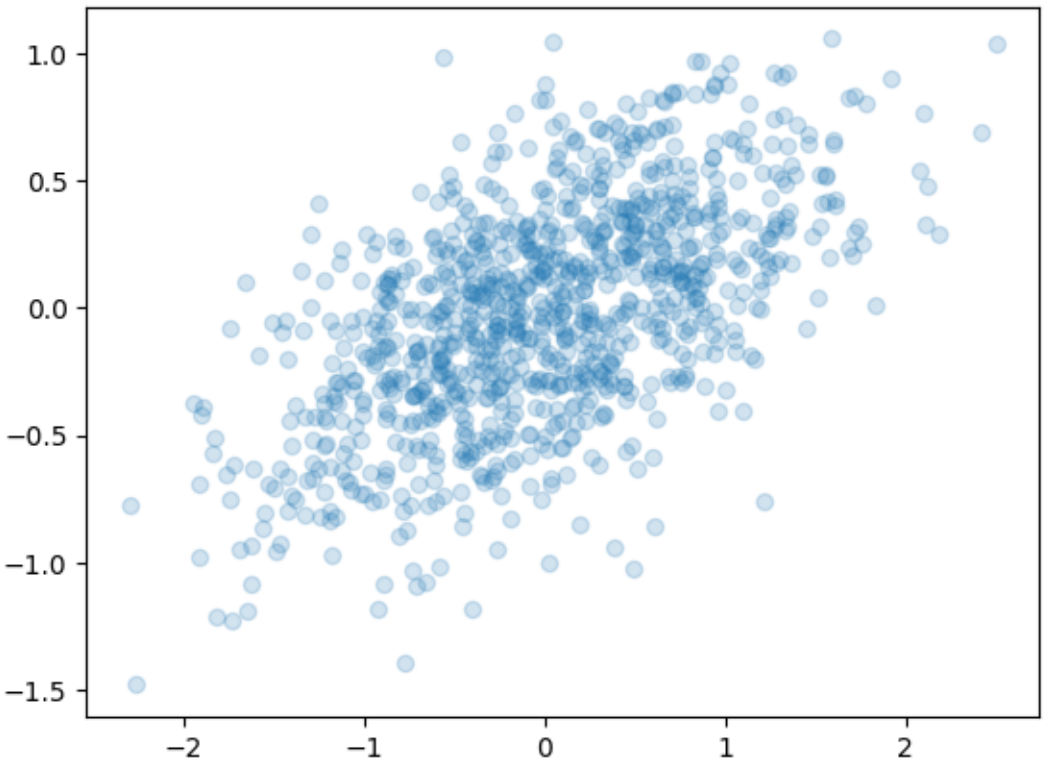
Even though the latter is usually preferred, let's have a look first at the former.

PCA is a particularv way of changing variables (== changing the base of the phase space) that is linear. It is not the only one, but typically performing only linear operations is sufficient for most datasets. Anyway, there are also cases where linear transformations are not enough and the true important variables are non linear combinations of the original variables. A classical example: say yuo are measuring the decay of a particle. You measure the momentum, i.e. the components px, py, pz, and the mass m, bu the variable you really care about is the invariant mass , which is a non linear combination of these. By plotting the components of the momentum separately, you would see nothing! In such cases where tecniques like PCA are insufficient, one usually uses machine learning techniques like autoencoders.

[8]:
```
# construct a dataset with a skewed 2D distribution
mu = [0,0] # <== zero mean! useful for later
sigma = [[0.6,0.2],[0.2,0.2]] # asymmetric sigmas
n = 1000
X = np.random.multivariate_normal(mu, sigma, n).T

plt.scatter(X[0,:], X[1,:], alpha=0.2)
```

[8]:
```
<matplotlib.collections.PathCollection at 0x117215c10>
```



in the basis of the covariance matrix, the variables are uncorrelated by definition (the transformed covariance matrix is diagonal)

the eigenvalue expresses the VARIANCE of the corrseponding new variable (after transformation). Therefore, ranking by magnitude of the eigenvalues mean that you can identify the variables that have the largest variance, and thus are the most relevant to your data analysis

[ ]:
```
# the covariance matrix
np.cov(X)
```

rememebr the "physical meaning": diagonal entries of the covariance matrix always show the variance along the basis vector direction

[13]:
```
# now find the eigenvectors of the covariance matrix..
#l, V = np.linalg.eig(np.cov(X))
l, V = la.eig(np.cov(X))

print (l, end= "\n\n")
print (V)

# First recall that V is an orthogonal matrix (and thus its transpose is also␣
 ↪its inverse)
V.dot(V.T)
```

```
[0.6926+0.j 0.1107+0.j]

[[ 0.9319 -0.3626]
 [ 0.3626  0.9319]]
```

[13]:
```
array([[1., 0.],
       [0., 1.]])
```

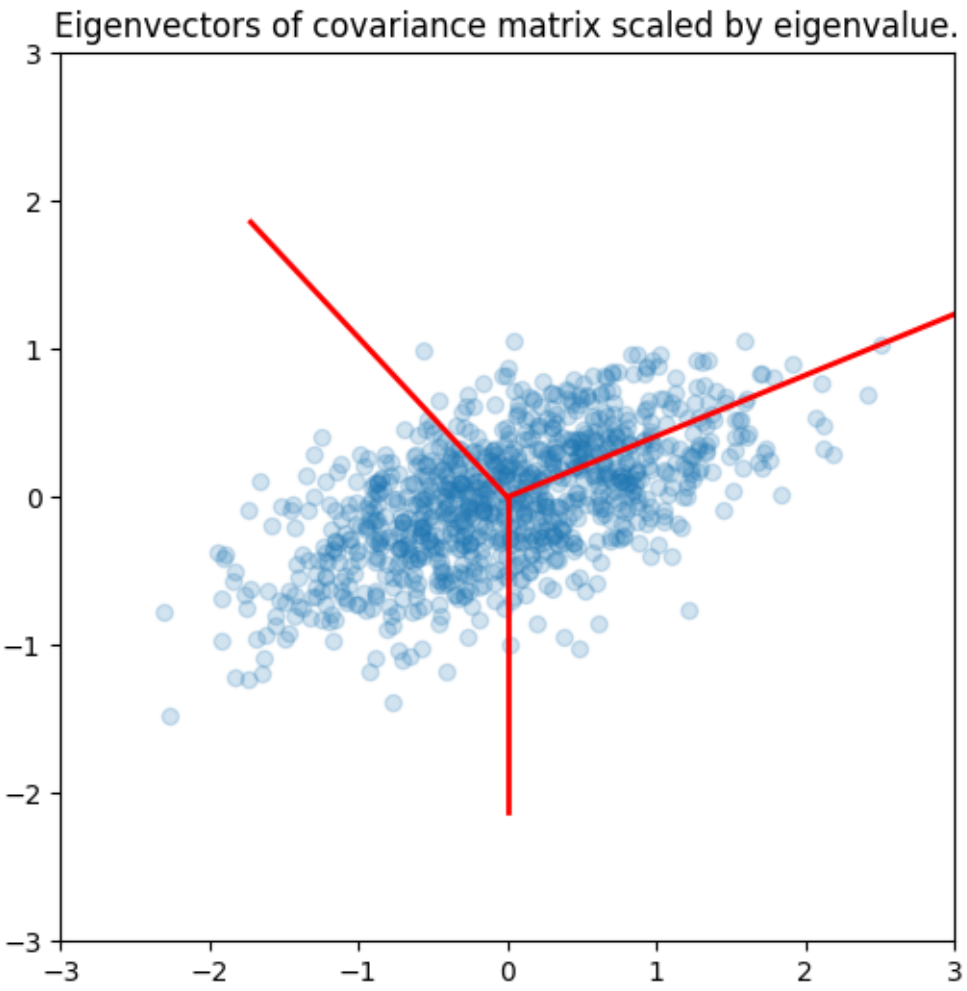
[11]:
```
plt.figure(figsize=(6,6))
# the original data distribution
plt.scatter(X[0,:], X[1,:], alpha=0.2)

# a scale factor to emphazise the lines
scale_factor=3

# draw each eigenvector
for li, vi in zip(l, V.T):
    print (li, vi)
    # the line is defined by means of its beginning and its end
    plt.plot([0, scale_factor*li*vi[0]], [0, scale_factor*li*vi[1]], 'r-', lw=2)

# fix the size of the axes to have the right visual effect
plt.axis([-3,3,-3,3])
plt.title('Eigenvectors of covariance matrix scaled by eigenvalue.');
```

```
(5.854101966249682+0j) [0.1802 0.0743 0.9808]
(-0.8541019662496852+0j) [ 0.6721 -0.7249  0.1509]
(1.0000000000000002+0j) [-0.     -0.7071  0.7071]
```



CHECK AGAIN

In the case the features of the datasets have all zero mean, the covariance matrix is of the form:

$$\text{Cov}(X) = \frac{XX^T}{n-1}$$

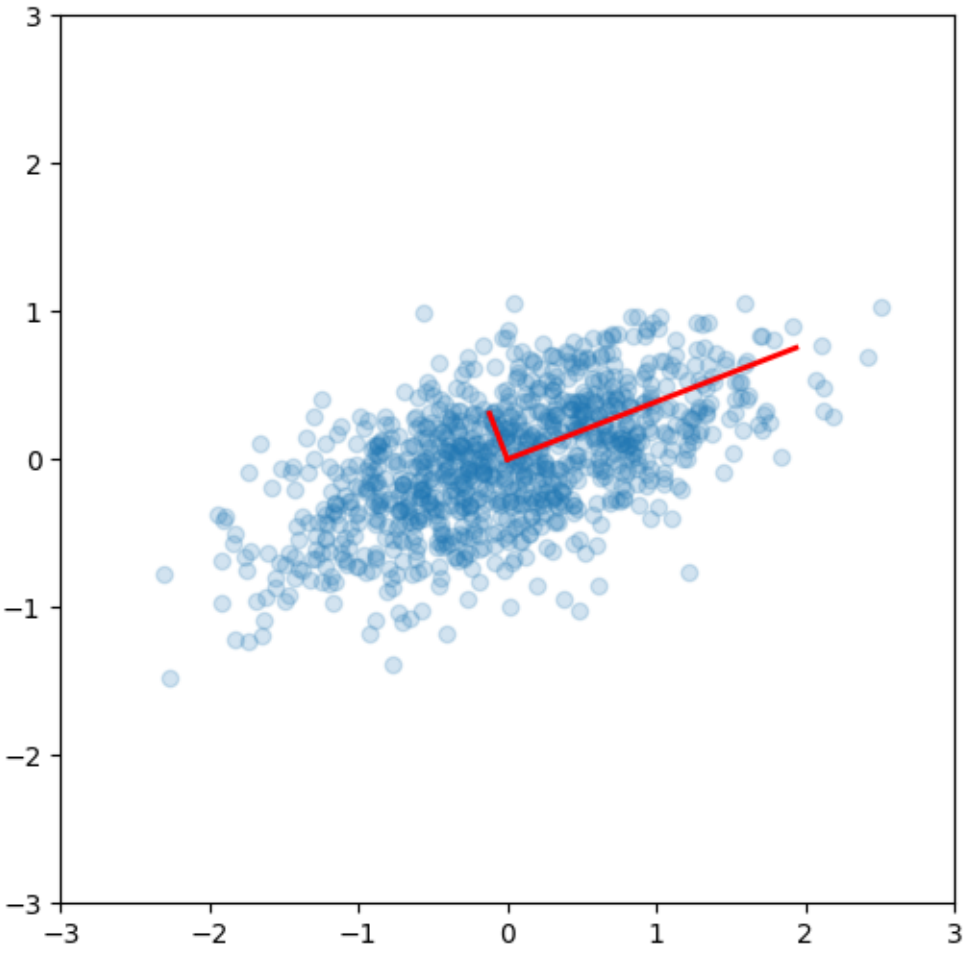
and thus

[15]:
```
l0, V0 = np.linalg.eig(np.dot(X, X.T)/(n-1))
print (l0)
print (V0)

plt.figure(figsize=(6, 6))
plt.scatter(X[0,:], X[1,:], alpha=0.2)
for li, vi in zip(l0, V0.T):
    plt.plot([0, scale_factor*li*vi[0]], [0, scale_factor*li*vi[1]], 'r-', lw=2)
```

```
plt.axis([-3,3,-3,3]);
```

```
[0.6926 0.1108]
[[ 0.9319 -0.3627]
 [ 0.3627  0.9319]]
```

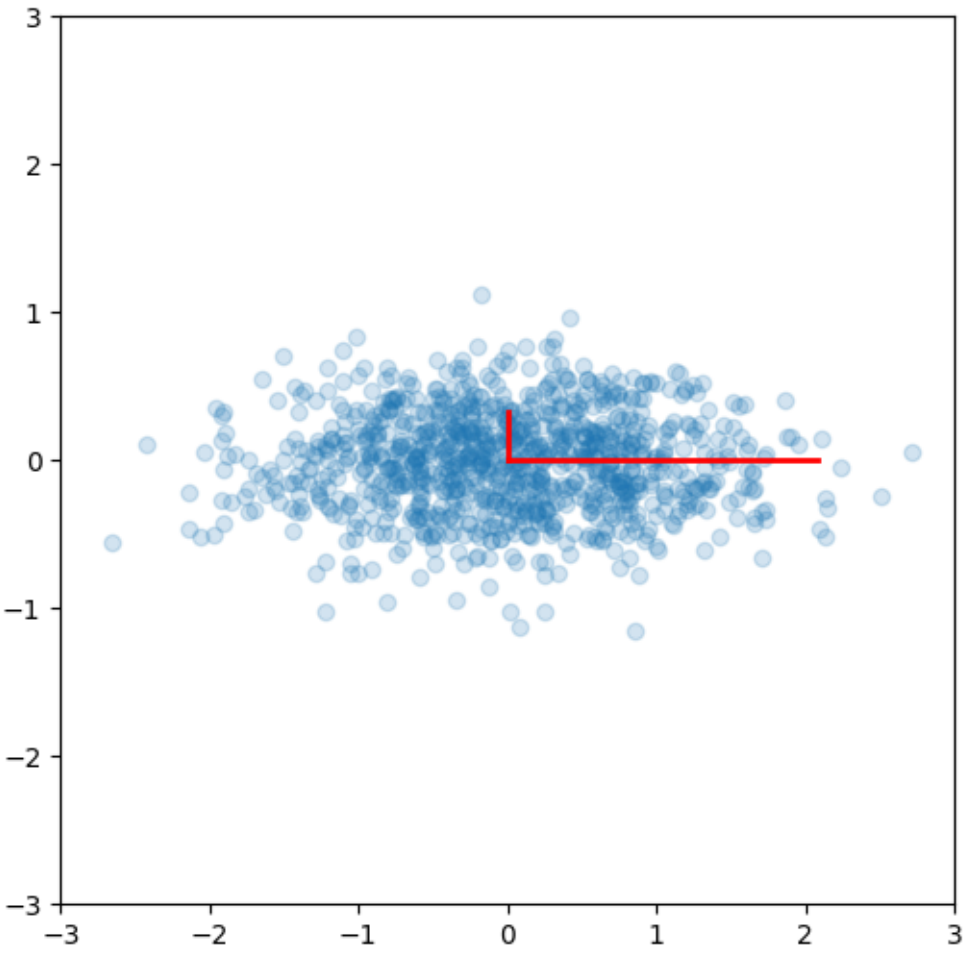


We can now use the eigenvectors and eigenvalues to rotate the data, i.e. **take the eigenvectors as new basis vectors and redefine the data points w.r.t this new basis.**

[16]:
```
# rotate all the data points accordingly to the new base
Xp = np.dot(V0.T, X)
```

[17]:
```
# then plot the rotated dataset and its "axes"
plt.figure(figsize=(6, 6))
plt.scatter(Xp[0,:], Xp[1,:], alpha=0.2)
# same eigenvalues as before, assume we rotated properly the data
for li, vi in zip(l0, np.diag([1]*2)):
```

```
    plt.plot([0, scale_factor*li*vi[0]], [0, scale_factor*li*vi[1]], 'r-', lw=2)
plt.axis([-3,3,-3,3]);
```



For example, if we only use the first column of xp, we will have the projection of the data onto the first principal component, **capturing the majority of the variance in the data with a single feature** that is a linear combination of the original features.

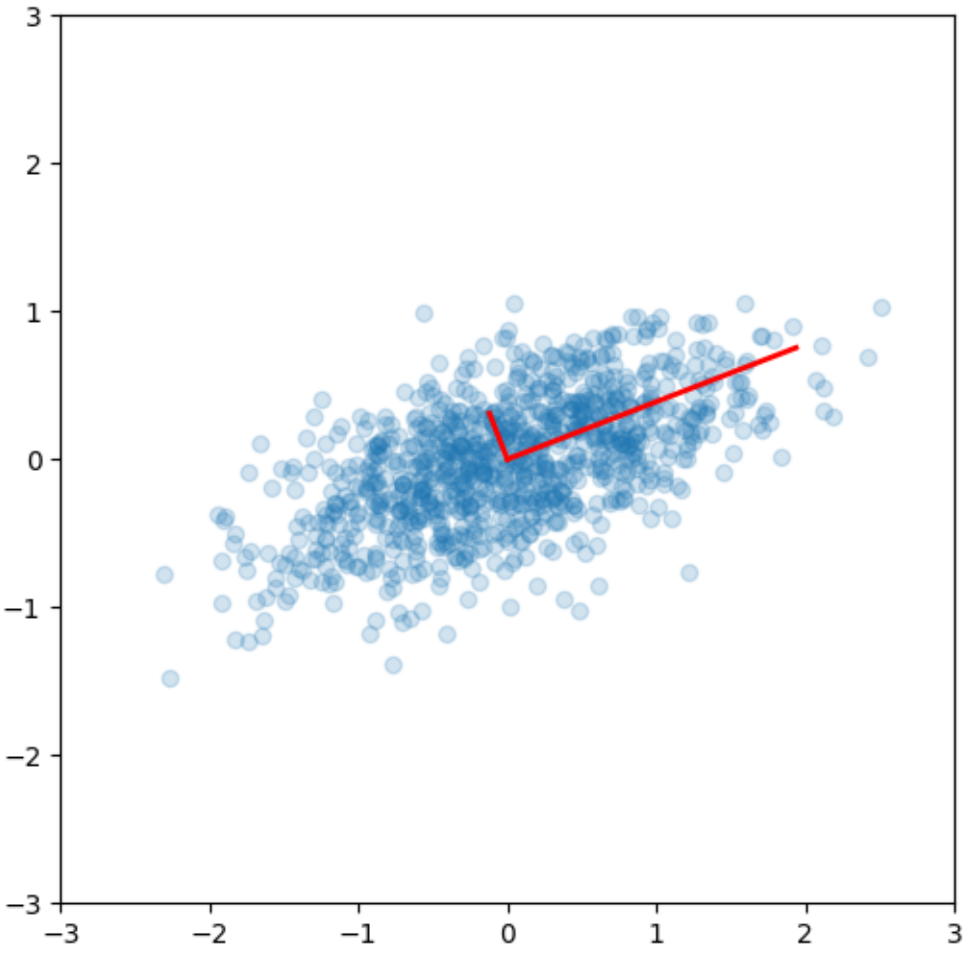
Of course you do not get rid of variables when you have only two! you do this kind of thing when you have a lot more like dozens or hundreds

We may need to transform the (reduced) data set to the original feature coordinates for interpretation. This is simply another linear transform (matrix multiplication).

[18]:
```
plt.figure(figsize=(6, 6))
Xpp = np.dot(V0, Xp)
plt.scatter(Xpp[0,:], Xpp[1,:], alpha=0.2)
for li, vi in zip(l0, V0.T):
    plt.plot([0, scale_factor*li*vi[0]], [0, scale_factor*li*vi[1]], 'r-', lw=2)
```

```
plt.axis([-3,3,-3,3])
```

[18]: (-3.0000, 3.0000, -3.0000, 3.0000)



1.5 Dimension reduction via PCA

Given the spectral decomposition:

$$A = V\Lambda V^{-1}$$

with Λ of rank p. Reducing the dimensionality to $k < p$ simply means setting to zero all but the first k diagonal values (ordered from the largest to the smaller in module; that is the default in numpy/scipy).

In this way we catch the most relevant part of its variability (covariance).

[ ]:
```
l, V = np.linalg.eig(np.cov(X))
Lambda=np.diag(l)
print (Lambda)
print ("A.trace():", np.cov(X).trace())
print ("Lambda.trace():", Lambda.trace())

print (Lambda[0,0]/Lambda.trace())
```

Since the **trace is invariant under change of basis**, the **total variability is also unchaged by PCA**.

By keeping only the first k principal components, we can still "explain" a fraction

$$\sum_{1}^{k} \lambda_i / \sum_{1}^{p} \lambda_i$$

of the total variability. Sometimes, the degree of dimension reduction is specified as keeping enough principal components so that (say) 90% fo the total variability is exlained.

rule of thunb: the more rows you have in the dataset, the better (because that reduces statistical uncertainty), whereas the more columns you have, the worst (because you dont know what to look at, so you use PCA)

as physisicsts, we want to deal with quantities that make sense, that we know what they are and have the same (physical) dimensions. PCA doesn't help in that, because it mixes things up in a way that we are left with quantities we cannot make sense of. Thus, PCA is ok but the second step of a good data analysis is typically to find a way to come back to quantities that make sense

1.6 SVD for PCA (??)

the two are equivalent up to a scale factor in the case where X data matrix has zero mean

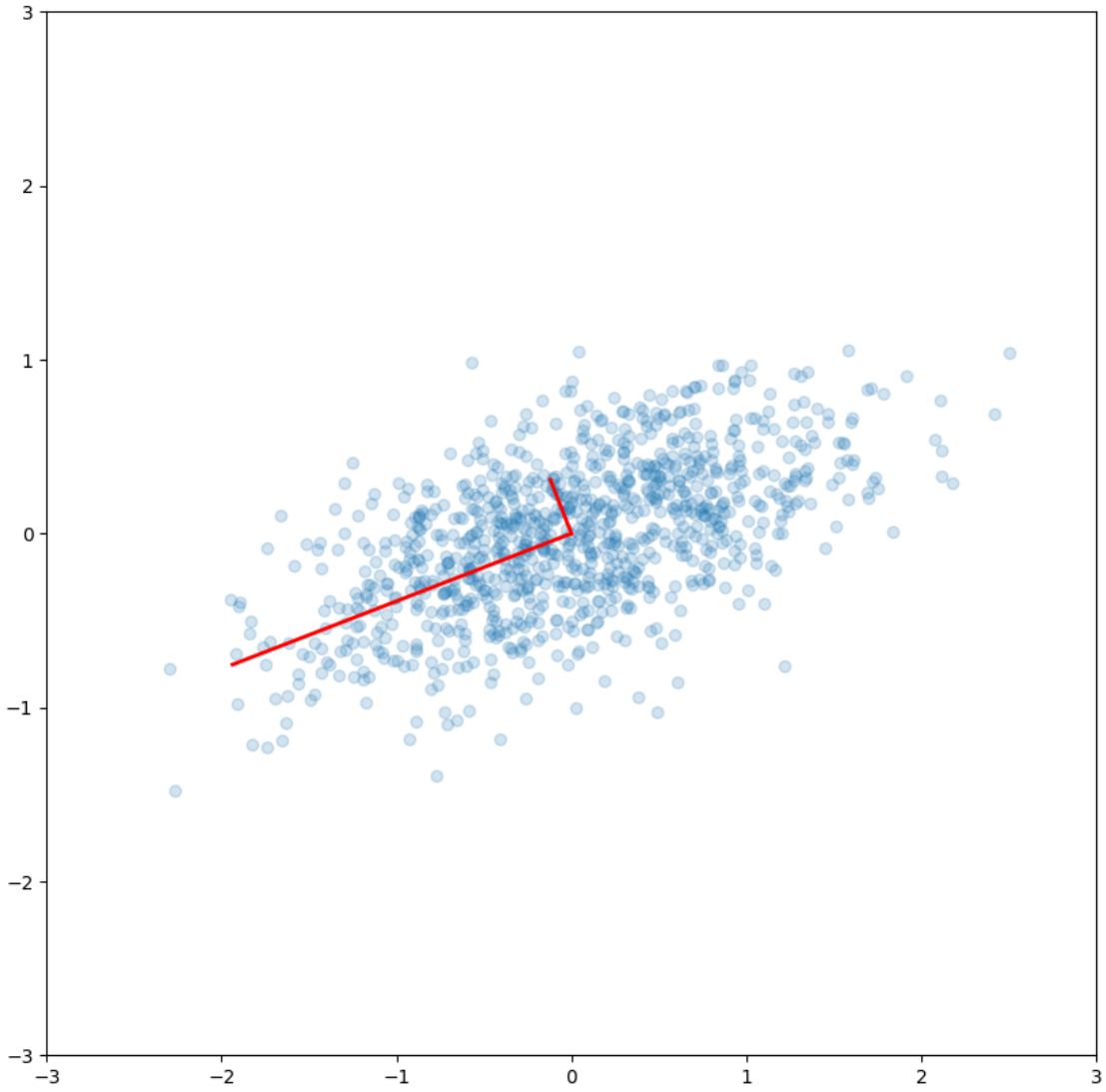
We saw that SVD is a decomposition of the data matrix $X = UDV^T$ where U and V are orthogonal matrices and D is a diagonal matrix.

Compare with the eigendecomposition of a matrix $A = W\Lambda W^{-1}$, we see that SVD gives us the eigendecomposition of the matrix XX^T, which as we have just seen, is basically a scaled version of the covariance **for a data matrix with zero mean**, with the eigenvectors given by U and eigenvalues by D^2 (scaled by n−1).

[19]:
```
U, spectrum, Vt = np.linalg.svd(X)

l_svd = spectrum**2/(n-1)
V_svd = U

plt.figure(figsize=(10,10))
plt.scatter(X[0,:], X[1,:], alpha=0.2)
for li, vi in zip(l_svd, V_svd):
    plt.plot([0, scale_factor*li*vi[0]], [0, scale_factor*li*vi[1]], 'r-', lw=2)
plt.axis([-3,3,-3,3]);
```



[20]:
```
print ("eigendecomposition:",l)
print ("SVD:",l_svd)
```

```
eigendecomposition: [0.6926+0.j 0.1107+0.j]
SVD: [0.6926 0.1108]
```