Goal this week: SVD, PCA (Review CS132 Geometric Algorithms)

Review linear Algebra.

Vectors and Matrices.

$$v = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_d \end{bmatrix}$$ → reals

is equivalent to a d-dimensional point ($\mathbb{R}^d$).

→ column vector.

$v^T = [v_1, ..., v_d]$ → row vector.

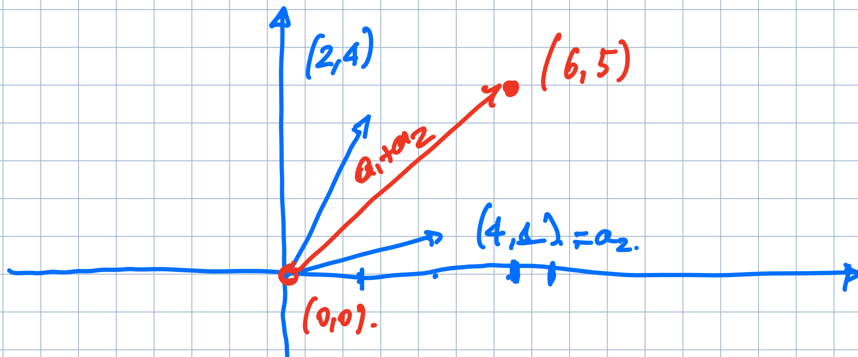
⟹ both will be stored as an array.

$$A^{n \times d} = \begin{bmatrix} - a_1^T - \\ \vdots \\ - a_n^T - \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1d} \\ \vdots & & & \\ A_{21} & A_{22} & \cdots & A_{2d} \\ & \vdots & & \\ A_{n1} & A_{n2} & \cdots & A_{nd} \end{bmatrix}$$

(a_1^T = first row; b_1 = first column)

$A_{i,j}$ — i: row index, j: col index.

$$A = \begin{bmatrix} | & & | \\ b_1 & \cdots & b_d \\ | & & | \end{bmatrix}$$



$a_1 + a_2 = (2+4, 4+1) =$

$= (6, 5)$.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} (1,2)\cdot\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \\ (3,4)\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \end{bmatrix} = \begin{bmatrix} x_1 + 2x_2 \\ 3x_1 + 4x_2 \end{bmatrix}$$

$A \cdot x.$

↳ dot product perspective of MATRIX VECTOR Multiplication.

"LINEAR COMBINATION of columns" perspective.

$$\begin{bmatrix} \begin{bmatrix} 1 \\ 3 \end{bmatrix} & \begin{bmatrix} 2 \\ 4 \end{bmatrix} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = x_1 \begin{bmatrix} 1 \\ 3 \end{bmatrix} + x_2 \begin{bmatrix} 2 \\ 4 \end{bmatrix} = \begin{bmatrix} x_1 \\ 3x_1 \end{bmatrix} + \begin{bmatrix} 2x_2 \\ 4x_2 \end{bmatrix} = \begin{bmatrix} x_1 + 2x_2 \\ 3x_1 + 4x_2 \end{bmatrix}$$

$$2x_1 + 5x_2 - 7x_3 = 10.$$

$$-x_1 + 8x_2 - 2x_3 = -2.$$

$$\underset{2\times 3}{A} \cdot \underset{3\times 1}{x} = \underset{2\times 1}{b}.$$

$$\underbrace{\begin{bmatrix} 2 & 5 & -7 \\ -1 & 8 & -2 \end{bmatrix}}_{A} \cdot \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}}_{x} = \begin{bmatrix} 10 \\ -2 \end{bmatrix}$$

$$A \cdot x = b.$$

we are trying to find a linear combination of the three 2dim columns of A such that this linear combination is equal to b.

$x, y \in \mathbb{R}^d$

$$x + y = \begin{bmatrix} x_1 \\ \vdots \\ x_d \end{bmatrix} + \begin{bmatrix} y_1 \\ \vdots \\ y_d \end{bmatrix} = \begin{bmatrix} x_1 + y_1 \\ \vdots \\ x_d + y_d \end{bmatrix}.$$

y, x+y, x

$$\overset{n\times d}{A} + \overset{n\times d}{B} = \overset{n\times d}{C} \quad \text{where} \quad C_{ij} = A_{ij} + B_{ij} \quad \forall \begin{matrix} 1 \le i \le n \\ 1 \le j \le d \end{matrix}$$

$$\overset{n\times m}{C} = \overset{n\times \bar{d}}{A} \cdot \overset{\bar{d}\times m}{B} \quad \text{where} \quad C_{ij} = \sum_{k=1}^{d} A_{ik} B_{kj} = a_i^T b_j$$

see here

$$\begin{bmatrix} - & a_1^T & - \\ & \vdots & \\ - & a_i^T & - \\ & \vdots & \\ - & a_n^T & - \end{bmatrix} \begin{bmatrix} | & | & | \\ b_1 \cdots & b_j & \cdots b_m \\ | & | & | \end{bmatrix}$$

$5 \cdot 3 = 3 \cdot 5 = 15.$

but $A \cdot B \neq B \cdot A$ (matrix multiplication is not commutative!).

$(AB)C = A(BC)$ (ASSOCIATIVE) ← but not commutative.

$A(B+C) = AB + AC$ (DISTRIBUTIVE)

VECTOR-VECTOR PRODUCT.

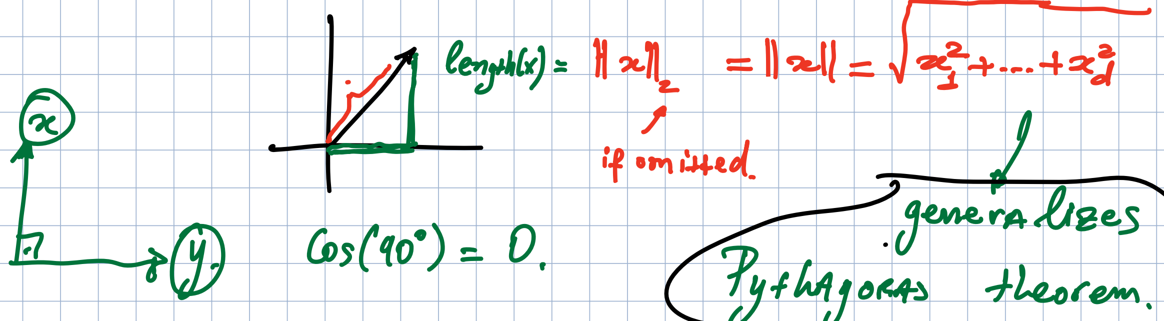
$$x^T y = \overset{1\times d}{(x_1 \ldots x_d)} \overset{d\times 1}{\begin{pmatrix} y_1 \\ \vdots \\ y_d \end{pmatrix}} = x_1 y_1 + \ldots + x_d y_d = \sum_{i=1}^{d} x_i y_i = \langle x, y \rangle$$

\langle

INNER PRODUCT

$$x^T \cdot y = \text{length}(x)\ \text{length}(y).\ \cos(\theta).\ \in \mathbb{R}.$$

→ A lot of geometry.



$\text{length}(x) = \|x\|_2 = \|x\| = \sqrt{x_1^2 + \ldots + x_d^2}$

if omitted.

generalizes Pythagoras theorem.

$$\left(\overset{d\times 1}{x}\ \overset{1\times k}{y^T}\right) = \begin{pmatrix} x_1 y_1 & \cdots & x_1 y_k \\ \vdots & x_i y_j & \vdots \\ x_d y_1 & \cdots & x_d y_k \end{pmatrix}_{d\times k} = \overset{d\times k}{C} \in \mathbb{R}^{d\times k}$$

OUTER Product

$x_i y_j$ → (i,j)-th entry.

$$\text{length}(x) = \|x\|_2 = \|x\| = \sqrt{x_1^2 + \dots + x_d^2} = \sqrt{\langle x, x\rangle} = \sqrt{x^T \cdot x}$$

($\|x\|_2$ ← if omitted)

Euclidean NORM. / ℓ_2.

$$\|x\|_2^2 = x^T \cdot x = \sum_{i=1}^{d} x_i^2$$

L_p norms (well-defined for our purposes for $p \in [1, +\infty)$).

$$\|x\|_p = \left(\sum_{i=1}^{d} |x_i|^p\right)^{1/p}$$

$\left[\|x\|_p^p = \sum_{i=1}^{d} |x_i|^p \right]$ — p (superscript) → POWER; p (subscript) denotes the norm

$$\|x\|_1 = \sum_{i=1}^{d} |x_i|$$

$$\|(1,-1)\|_1 = |1| + |-1| = 2$$

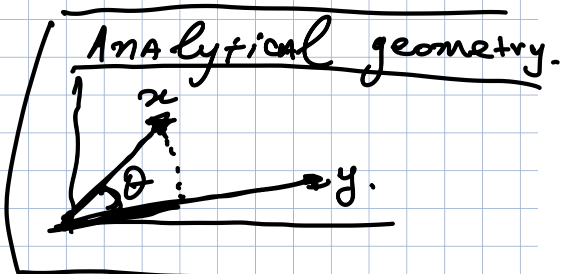
$$\|x\|_\infty = \max_{1\le i\le d} |x_i|$$

$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \quad y = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$

Q. $x^T y = x_1 y_1 + x_2 y_2$

$$\|x\| = \sqrt{x_1^2 + x_2^2} \qquad \|y\| = \sqrt{y_1^2 + y_2^2}$$

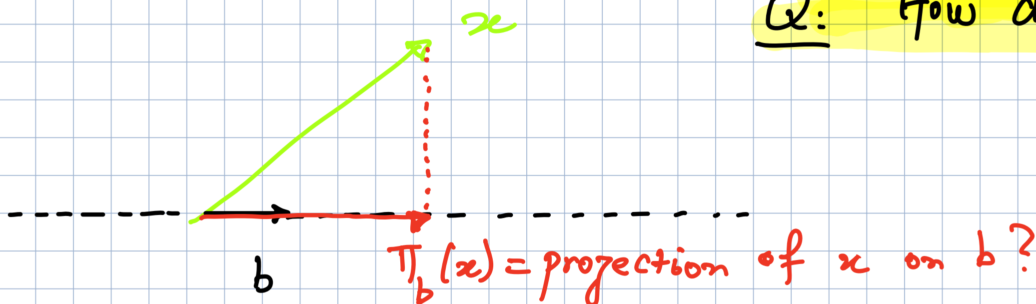
Analytical geometry.



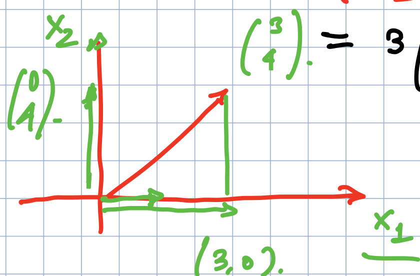
$$\cos\theta = \frac{x_1 y_1 + x_2 y_2}{\sqrt{x_1^2+x_2^2}\,\sqrt{y_1^2+y_2^2}} \qquad \text{(projections)}$$

How do we project (why does trigonometry play a role?).

Q: How do we project x on b?



Example: $x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$, $b = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$. then $\pi_b(x) = \,? = \begin{pmatrix} x_1 \\ 0 \end{pmatrix}$.



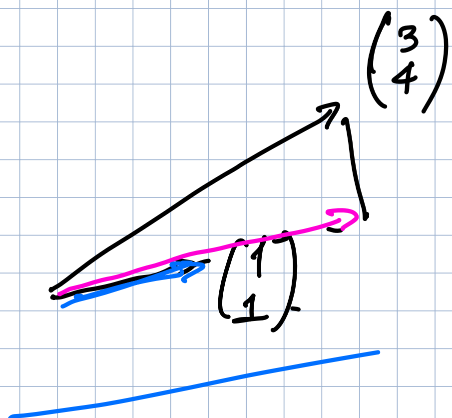
$$\begin{pmatrix} 3 \\ 4 \end{pmatrix} = 3\begin{pmatrix} 1 \\ 0 \end{pmatrix} + 4\begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

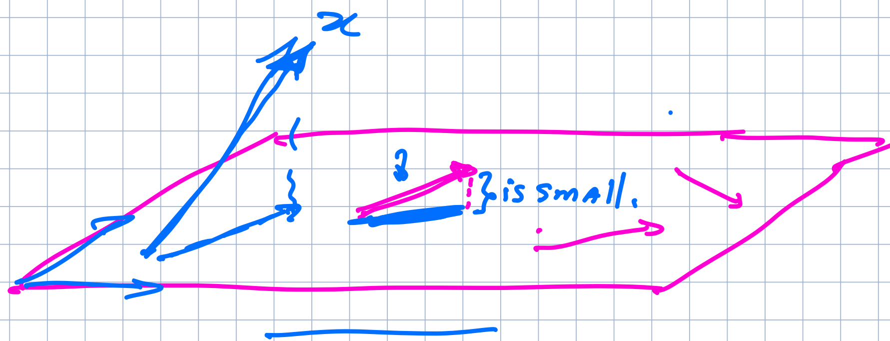
$$\pi_{\begin{pmatrix} 1 \\ 0 \end{pmatrix}}\left(\begin{pmatrix} 3 \\ 4 \end{pmatrix}\right) = \begin{pmatrix} 3 \\ 0 \end{pmatrix}.$$

$$\pi_{\begin{pmatrix} 0 \\ 1 \end{pmatrix}}\left(\begin{pmatrix} 3 \\ 4 \end{pmatrix}\right) = \begin{pmatrix} 0 \\ 4 \end{pmatrix}.$$

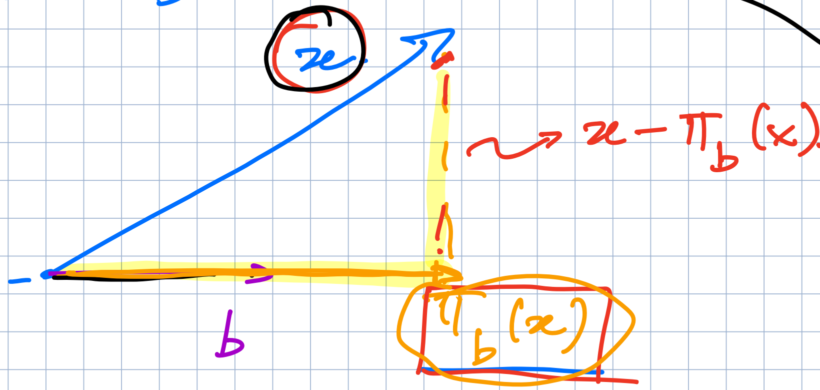
This example is easy, because $b = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix}$ STANDARD ORTHONORMAL basis for $\mathbb{R}^2$.

($e_1 = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$, $e_2 = \begin{pmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$, ----, $e_n = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix}$ is The STANDARD BASIS for $\mathbb{R}^n$).





Clearly. $\pi_b(x) = \lambda \cdot b$ for some $\lambda \in \mathbb{R}$.



$x - \pi_b(x) \perp b$

$(x - \pi_b(x))^T b = 0 \Rightarrow x^T b = (\pi_b(x))^T b$

$(x - \pi_b(x))^T = (x^T - \pi_b^T(x))$

$\lambda b^T b = x^T b \Rightarrow \lambda = \frac{x^T \cdot b}{b^T b} = \frac{b^T x}{\|b\|_2^2}$

So $\underset{d\times 1}{\pi_b(x)} = \lambda \cdot b = \left(\frac{b^T x}{b^T b}\right) \cdot b = \underset{d\times d}{\left[\frac{(b\, b^T)}{(b^T b)}\right]} \underset{d\times 1}{x}$

Projection MATRIX.

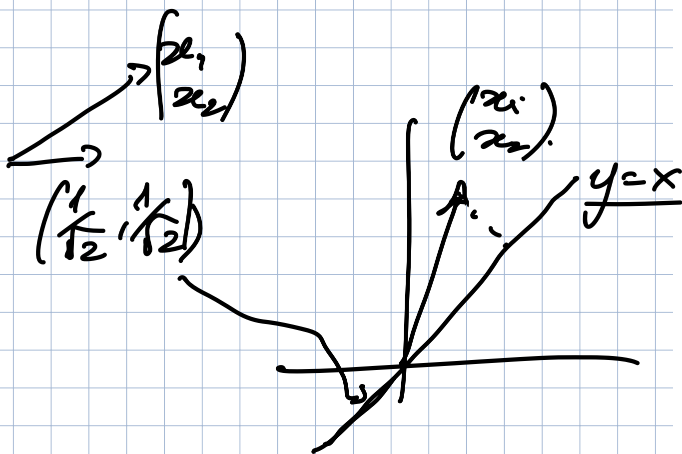
$P = \frac{b\, b^T}{b^T b}$

$P^2 = \left(\frac{b\, b^T}{b^T b}\right)\left(\frac{b b^T}{b^T b}\right) = \frac{b (b^T b) b^T}{(b^T b)^2} = \frac{b\, b^T}{b^T b} = P$

$$P^2 = P.$$

$$x^T b = x_1 b_1 + \dots + x_d b_d = b^T x. \checkmark$$

$$b = \left(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right).$$



$$b^T b = \frac{1}{2} + \frac{1}{2} = 1.$$

$$b b^T = \begin{pmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{pmatrix} \begin{pmatrix} 1/\sqrt{2} & 1/\sqrt{2} \end{pmatrix} = \begin{pmatrix} 1/2 & 1/2 \\ 1/2 & 1/2 \end{pmatrix}.$$

$$P = \begin{pmatrix} 1/2 & 1/2 \\ 1/2 & 1/2 \end{pmatrix}.$$

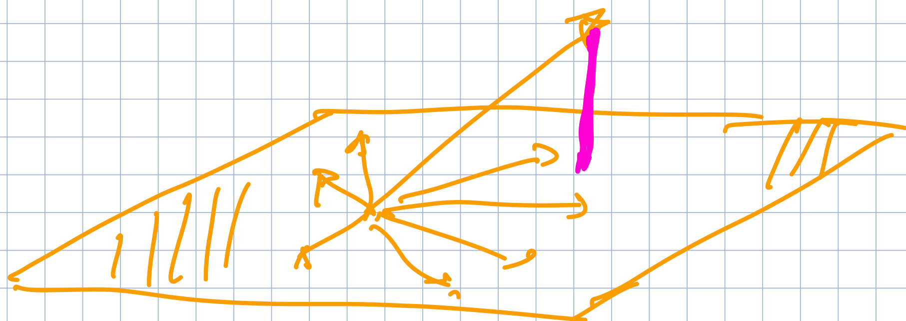
$$Px = \begin{pmatrix} \frac{1}{2} x_1 + \frac{1}{2} x_2 \\ \frac{1}{2} x_1 + \frac{1}{2} x_2 \end{pmatrix}$$

$$P^2 = P.$$

$$\frac{b b^T}{b^T b}$$

$$(Px) = \Pi_b(x)$$

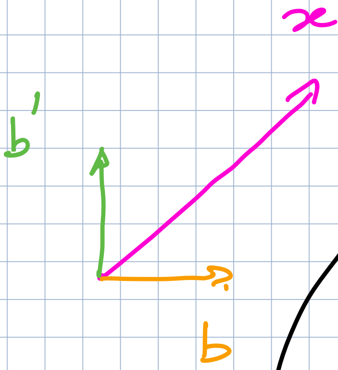
$$\Pi_b(\Pi_b(x)) = \Pi_b(x).$$



$\{x_1, \dots, x_n\}$

$\mathbb{R}^3$

idea 1 find b'. (e.g. if $b = \begin{pmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{pmatrix}$, then $b' = \begin{pmatrix} 1/\sqrt{2} \\ -1/\sqrt{2} \end{pmatrix}$). AND repeat.

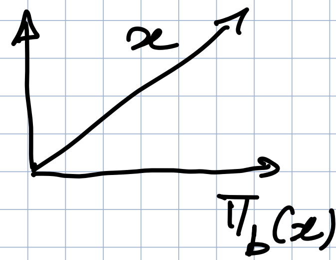


$$P' = \frac{b' b'^T}{\| (b')^T b'}$$

$$\pi_{b'}(x) = P' \cdot x.$$

idea 2 $(I - P)x = x - Px = x - \pi_b(x)$

$=$



Claim $P' = I - P$. (proof by example).

$$P = \begin{pmatrix} 1/2 & 1/2 \\ 1/2 & 1/2 \end{pmatrix} \quad \left(\text{on } b = \begin{pmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{pmatrix} \right).$$

$$I - P = \begin{pmatrix} 1/2 & -1/2 \\ -1/2 & 1/2 \end{pmatrix}. \quad \checkmark$$

$$\frac{(b')(b')^T}{1} =$$

$1 = (b')^T b'.$

$$\left(\left\| \left(\tfrac{1}{\sqrt{2}}, -\tfrac{1}{\sqrt{2}} \right) \right\| = \sqrt{\left(\tfrac{1}{\sqrt{2}}\right)^2 + \left(-\tfrac{1}{\sqrt{2}}\right)^2} = 1 \right.$$

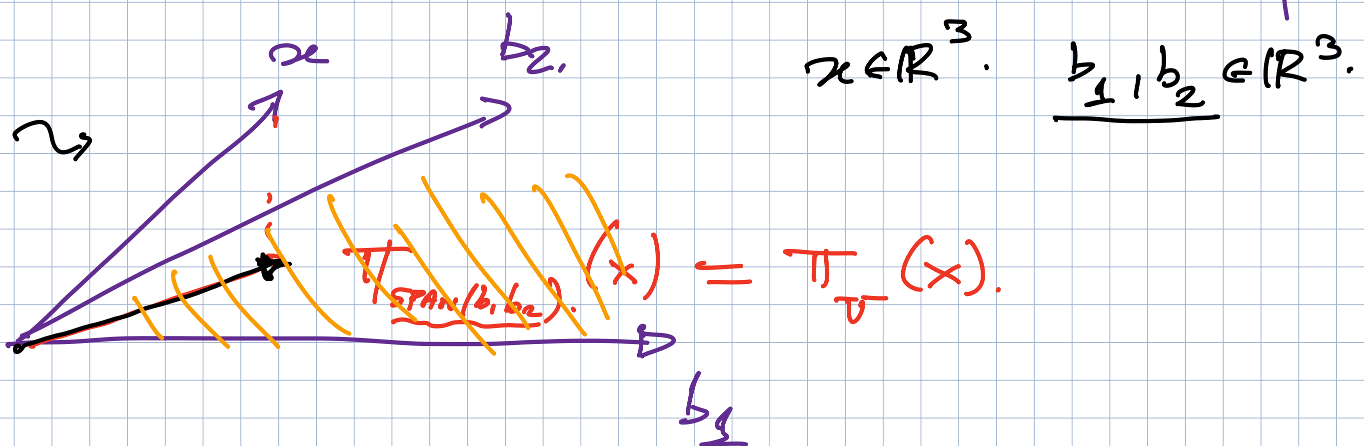
$$(I-P)^2 = (I-P)(I-P) = I - 2P + P^2$$

$$= I - 2P + P = I - P.$$

(Projection property. $X^2 = X$.).

Projection on general spaces.

$x \in \mathbb{R}^3$. $b_1, b_2 \in \mathbb{R}^3$.



$\pi_{span(b_1,b_2)}(x) = \pi_U(x).$

$U = span(b_1, b_2)$ (span $(b_1, \ldots, b_m)$ more generally.).

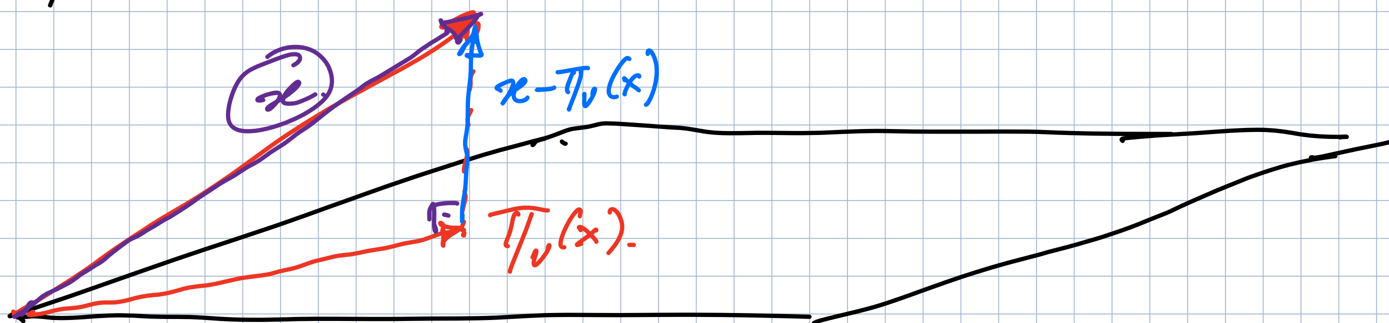
$$\pi_U(x) = \lambda_1 b_1 + \cdots + \lambda_m b_m \quad \text{for some } \lambda_1 \ldots \lambda_m.$$

$$= \underset{d \times m}{\begin{bmatrix} | & & | \\ b_1 & \cdots & b_m \\ | & & | \end{bmatrix}} \underset{m \times 1}{\begin{bmatrix} \lambda_1 \\ \vdots \\ \lambda_m \end{bmatrix}} = B \cdot \lambda$$

LECTURE 19. (APRIL 6TH, 2023)

Cont. from last time.

We are given a set of vectors $b_1, \dots, b_m$ (linearly independent) that span a subspace $U = \text{span}(b_1, \dots, b_m) \subseteq \mathbb{R}^n$



How do we project x on U? $\exists \lambda_1, \dots, \lambda_m \in \mathbb{R}$. such that

$\pi_U(x) = \lambda_1 b_1 + \dots + \lambda_m b_m$ or equivalently

$$\underbrace{\begin{bmatrix} | & & | \\ b_1 & \cdots & b_m \\ | & & | \end{bmatrix}}_{B} \underbrace{\begin{bmatrix} \lambda_1 \\ \vdots \\ \lambda_m \end{bmatrix}}_{\lambda} = \pi_U(x) \in \mathbb{R}^n.$$

Now we express the fact that $x - \pi_U(x) \perp b_i$, $i = 1 \dots m$.

$$\left.\begin{aligned} b_1^T (x - \pi_U(x)) &= 0 \\ &\vdots \\ b_m^T (x - \pi_U(x)) &= 0 \end{aligned}\right\} \Rightarrow \begin{bmatrix} - b_1^T - \\ \vdots \\ - b_m^T - \end{bmatrix} (x - B\lambda) = 0.$$

Notice $\begin{bmatrix} - b_1^T - \\ \vdots \\ - b_m^T - \end{bmatrix} = \begin{bmatrix} | & & | \\ b_1 & \cdots & b_m \\ | & & | \end{bmatrix}^T$ So we can rewrite:

why is it invertible

$$B^T x = (B^T B)\lambda \Rightarrow \lambda = (B^T B)^{-1} B x.$$

So our projection is $\pi_v(x) = B\lambda = B(B^T B)^{-1} B^T x$.

The projection matrix. $P = B(B^T B)^{-1} B^T$

Example if $b_1, \dots, b_m$ is orthonormal. $(B^T B = I)$ then

$P = B B^T$

In this case where $b_i^T b_j = \begin{cases} 1 & i=j \\ 0 & i \neq j \end{cases}$, this is equiv.

to $\pi_v(x) = B B^T x. = \begin{bmatrix} | & & | \\ b_1 & \cdots & b_m \\ | & & | \end{bmatrix} \begin{bmatrix} - b_1^T - \\ \vdots \\ - b_m^T - \end{bmatrix} = \begin{bmatrix} | & & | \\ b_1 & \cdots & b_m \\ | & & | \end{bmatrix} \begin{bmatrix} b_1^T x \\ \vdots \\ b_m^T x \end{bmatrix}$

$$= (b_1^T x) b_1 + \cdots + (b_m^T x) b_m.$$

Recall from previous lecture this is the projection of x on the 1dim subspace spanned by b_1. etc...

Reminder (important fact) Section 12.8 (online book. Blum - Hopcroft - Kannan).

Eigenvalue decomposition for symmetric ($A=A^T$) real matrices

THEOREM. Let A be a real symmetric matrix. Then,

1) The eigenvalues $\lambda_1, \dots, \lambda_n$ are real, as are the components of the corresponding eigenvectors $v_1, \dots, v_n$.

2) A is orthogonally diagonalizable,

$$A = \sum_{i=1}^{n} \lambda_i \, v_i v_i^T. \qquad \left(v_i^T v_j = \begin{cases} 1 & \text{if } i=j \\ 0 & \text{if } i \neq j \end{cases} \right).$$

Equivalently, $A = V D V^T$ where $D = \begin{bmatrix} \lambda_1 & & 0 \\ & \ddots & \\ 0 & & \lambda_n \end{bmatrix}$

and $V = \begin{bmatrix} | & & | \\ v_1 & \cdots & v_n \\ | & & | \end{bmatrix}$, $(V^T = V^{-1})$. ∎

Theorem A real matrix A is orthogonally diagonalizable if and only if A is symmetric.

Singular Value Decomposition (SVD)

For any matrix $A \in \mathbb{R}^{m \times n}$ there exist orthonormal matrices $U \in \mathbb{R}^{m \times m}$, $V \in \mathbb{R}^{n \times n}$ and a diagonal matrix

$$\Sigma = \left(\begin{array}{ccc|ccc} \sigma_1 & & & \cdots & 0 & \cdots\ 0 \\ & \ddots & & & & \\ & & \sigma_r & & & \\ \hline & & & 0 & & \\ & & & & \ddots & \\ & & & & 0 & 0 \cdots 0 \end{array} \right)^{m \times n} \quad (\text{if } m < n)$$

$$\Sigma = \begin{bmatrix} \sigma_1 & & & \\ & \ddots & & \\ & & \sigma_r & \\ & & & \ddots & \\ & & & & 0 \end{bmatrix} \quad \text{if } m = n.$$

$$\Sigma = \begin{bmatrix} \sigma_1 & & & & \\ & \ddots & & & \\ & & \sigma_r & & \\ \hline & & & 0 & \\ & & & & \ddots \\ 0 & & & & 0 \\ \vdots & & & & \vdots \\ 0 & & & & 0 \end{bmatrix} \quad \text{if } m > n.$$

With diagonal entries $\underbrace{\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r}_{\text{singular values}} > \sigma_{r+1} = \cdots = \sigma_{\min(m,n)} = 0$ ($r = \text{rank}(A)$)

such that $A = U \Sigma V^T = \sum_{i=1}^{r} \sigma_i u_i v_i^T$.

(U: left singular vectors; Σ: singular values; V^T: right singular vectors)

LEMMA (Analog of eigenvalues and eigenvector)

$$A v_i = \sigma_i u_i \quad (1) \quad \text{and} \quad A^T u_i = \sigma_i v_i. \quad (2)$$

PROOF

Since V is orthogonal, $A \cdot V = U \Sigma V^T V = U \Sigma$ which gives equation (1)

$$A^T u_i = \left(\sum_{k=1}^{r} \sigma_k u_k v_k^T\right)^T u_i = \left(\sum_{k=1}^{r} \sigma_k v_k u_k^T\right) u_i = \sigma_i v_i \underbrace{(u_i^T u_i)}_{1} = \sigma_i v_i$$

BEST k-RANK APPROX.

Define $A_k = \sum_{i=1}^{k} \sigma_i u_i v_i^T \quad (k = 1, 2, \ldots, r).$

LEMMA The rows of A_k are the projections of the rows of A

onto the subspace.

$$A^{m\times n} = \begin{bmatrix} | & & | & | & & | \\ u_1 & \cdots & u_r & u_{r+1} & \cdots & u_m \\ | & & | & | & & | \end{bmatrix} \cdot \Sigma \cdot \begin{bmatrix} - & v_1^T & - \\ & \vdots & \\ - & v_r^T & - \\ - & v_{r+1}^T & - \\ & \vdots & \\ - & v_n^T & - \end{bmatrix}$$

$\text{rank}(A) = r \leq \min(m,n)$. → when equality holds, A is full rank.

$R(A) = R([u_1, \ldots, u_r])$. $N(A^T) = R([u_{r+1}, \ldots, u_m])$.

$R(A^T) = R([v_1, \ldots, v_r])$. $N(A) = R([v_{r+1}, \ldots, v_n])$.

The 2-norm of a matrix. $\|A\|_2 \triangleq \sup_x \dfrac{\|Ax\|_2}{\|x\|_2} = \sigma_1$. → give geom interpretation

The Frobenius norm $\|A\|_F^2 \triangleq \sum_{i=1}^{m}\sum_{j=1}^{n} a_{ij}^2 = \sigma_1^2 + \cdots + \sigma_r^2$.

Theorem For any k-rank matrix B: $\|A - A_k\|_F \leq \|A - B\|_F$,

$\|A - A_k\|_2 \leq \|A - B\|_2$. Furthermore, $\|A - A_k\|_2 = \sigma_{k+1}^2$.

Important Remark

$$A^TA = V\Sigma^T\Sigma V^T = V\Sigma^2V^T.$$
$$AA^T = U\Sigma^2U^T.$$

Thus σ_i^2 are the eigenvalues of A^TA, AA^T and $\{v_i\}$, $\{u_i\}$ are the eigenvectors respectively.

LEAST SQUARES VIA SVD

$$\min_x \|Ax - b\|_2^2$$

$$\|Ax-b\|_2^2 = \|(U\Sigma V^T)x - b\|_2^2 = \|U^T(AVV^Tx - b)\|_2^2 =$$

$$= \|\Sigma(V^Tx) - U^Tb\|_2^2 = \|\Sigma z - U^Tb\|_2^2$$

$$= \sum_{i=1}^{r} (\sigma_i z_i - u_i^T b)^2 + \sum_{i=r+1}^{m} (u_i^T b)^2$$

The opt solution is given by $z_i = \dfrac{u_i^T b}{\sigma_i}$. $i=1\dots r$.

AND the obj. becomes $\displaystyle\sum_{i=r+1}^{m} (u_i^T b)^2 = \min_x \|Ax-b\|_2^2$.

Let's find the actual x^*:

$$x^* = V z^* \Rightarrow \boxed{x^* = \sum_{i=1}^{r} \left(\frac{u_i^T b}{\sigma_i}\right) v_i}$$

Lecture 19

CS365 Foundations of Data Science, Prof. Tsourakakis

Today we will explore the Singular Value Decomposition (SVD) and Principal Component Analysis (PCA) that you have already seen in the prerequisite class CS132 Geometric Algorithms. The assigned readings are the following materials:

Readings

- Sections 3.1 to 3.6 and Section 12.8 from the Foundations of Data Science book.
- Class materials and notes, including this notebook.

PCA and its relation to SVD

Recall that Singular Value Decomposition (SVD) is a technique used to decompose a matrix into three simpler matrices. This technique is widely used in data science, machine learning, and image processing. The three matrices that make up the decomposition are:

- U: a matrix containing the left singular vectors of the original matrix.
- S: a diagonal matrix containing the singular values of the original matrix.
- V^T: a matrix containing the right singular vectors of the original matrix.

SVD is useful for reducing the dimensionality of a dataset, for example, to perform principal component analysis (PCA). It is also used in recommendation systems, image compression, and signal processing. We will see more examples later.

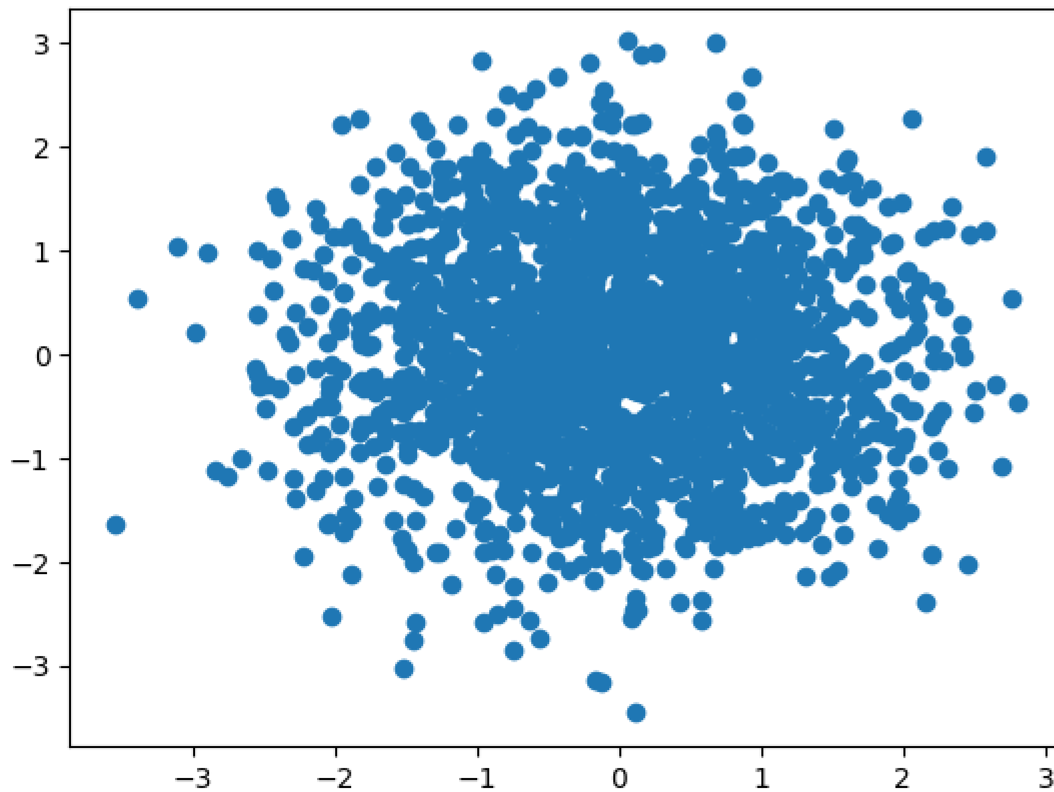
In [ ]:
```
import numpy as np
import random
import matplotlib.pyplot as plt
import cv2
from sklearn.decomposition import PCA
```

In [ ]:
```
# Mersenne Twister pseudo-random number generator.
rng = np.random.RandomState(None)
```

Let's generate a random cloud of 2000 2d points, and visualize it as follows.

In [ ]:
```
Y = rng.randn(2, 2000)
plt.scatter(Y[0, :], Y[1, :])
```

Out[ ]: <matplotlib.collections.PathCollection at 0x7fbeec997d30>



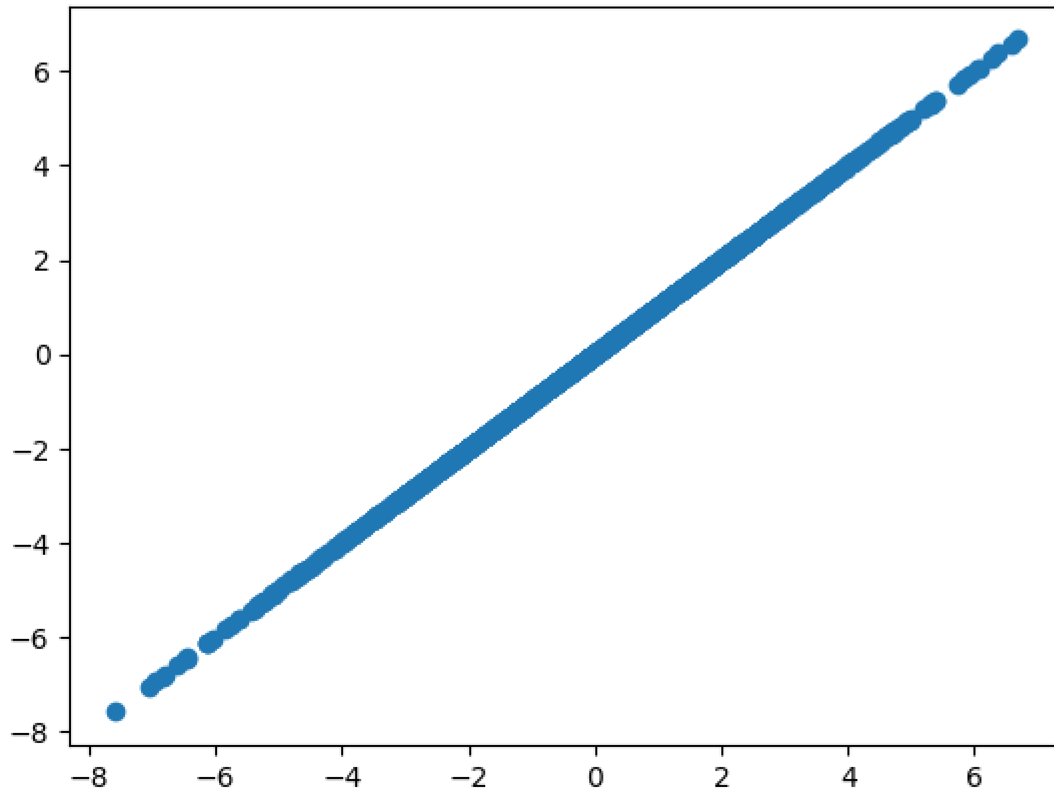
In [ ]:
```
np.corrcoef(Y)
```

Out[ ]:
```
array([[ 1.        , -0.01506826],
       [-0.01506826,  1.        ]])
```

Let's perform a special linear transformation, create a new dataset X1, and visualize it. What do you observe and why? Why do all points line now on a line?

In [ ]:
```
X1 = np.dot(np.array([[1, 2], [1, 2]]), Y)
plt.scatter(X1[0, :], X1[1, :])
```

Out[ ]: <matplotlib.collections.PathCollection at 0x7fbeea843850>

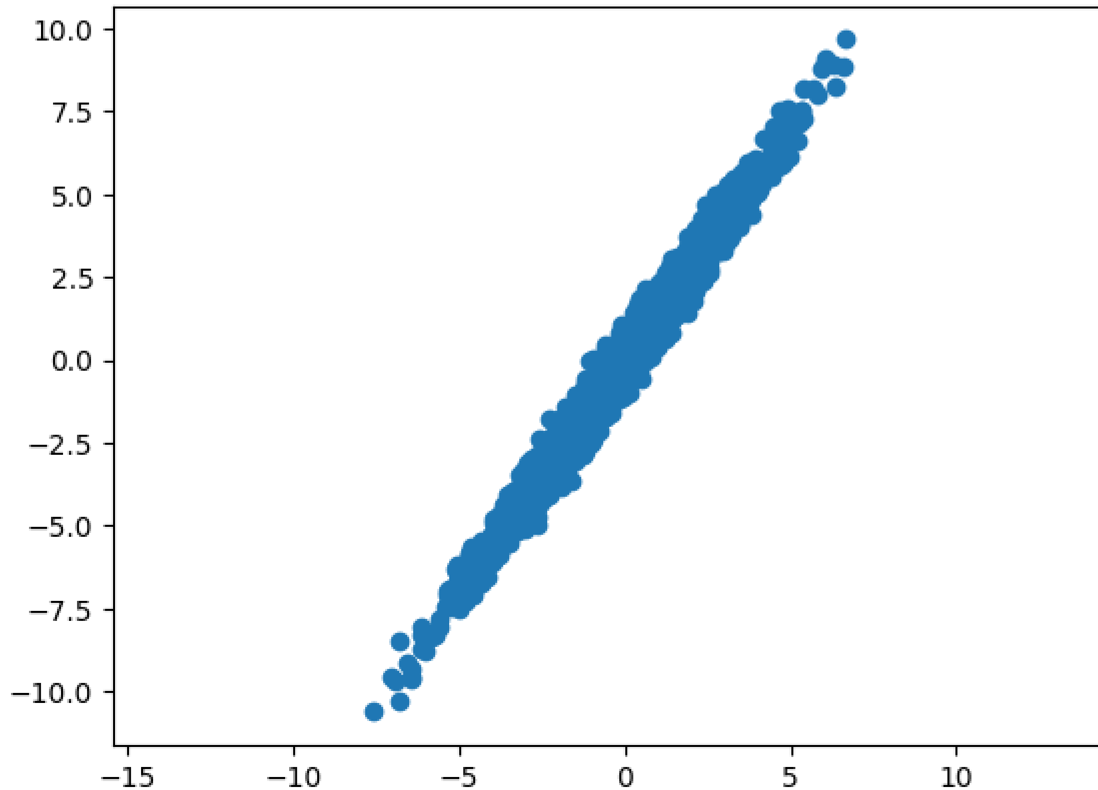


In [ ]:
```
np.corrcoef(X1)
```

Out[ ]:
```
array([[1., 1.],
       [1., 1.]])
```

Let's create a slightly different mapping, but still make sure most of the data variance is along one of the two dimensions.

In [ ]:
```
X = np.dot(np.array([[1, 2], [1, 3]]), Y)
plt.scatter(X[0, :], X[1, :])
plt.axis('equal');
```



In [ ]:
```
np.corrcoef(X)
```

Out[ ]:
```
array([[1.        , 0.98960585],
       [0.98960585, 1.        ]])
```

Question Explain why you observe the above transformations and values for corrcoeff.

In [ ]:
```
pca = PCA(n_components=2)
pca.fit(X.T)
```

Out[ ]:
```
▾ PCA
PCA(n_components=2)
```

In [ ]:
```
print(pca.components_)
```

```
[[-0.57575738 -0.8176206 ]
 [-0.8176206   0.57575738]]
```

In [ ]:
```
print(pca.explained_variance_/pca.explained_variance_.sum())
```

```
[0.99538503 0.00461497]
```

This information is encoded in the singular values. Specifically:

In [ ]:
```
np.square(pca.singular_values_)/(np.square(pca.singular_values_).sum())
```

Out[ ]:
```
array([0.99538503, 0.00461497])
```

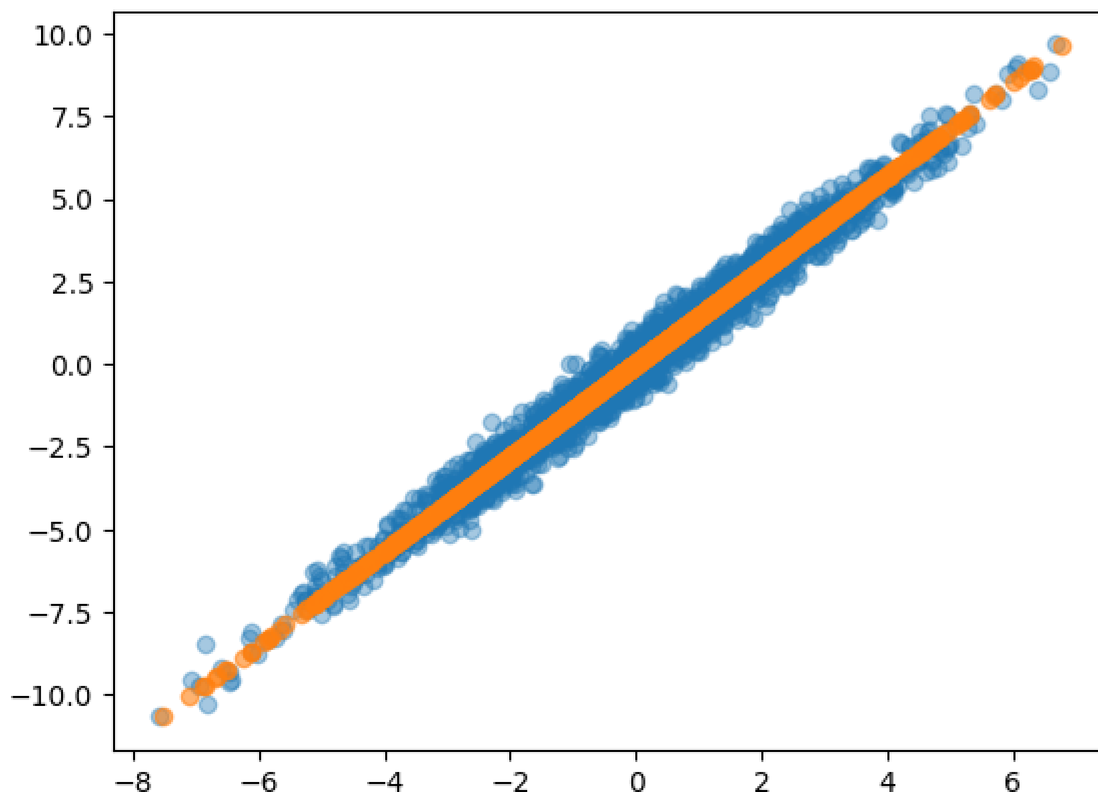
Let's project the data into the top principal component (PC).

In [ ]:
```
pca = PCA(n_components=1)
pca.fit(X.T)
X_pca = pca.transform(X.T)
print("original shape:   ", X.T.shape)
print("transformed shape:", X_pca.shape)
```

```
original shape:    (2000, 2)
transformed shape: (2000, 1)
```

In [ ]:
```
X_proj = pca.inverse_transform(X_pca)
X = X.T
plt.scatter(X[:, 0], X[:, 1], alpha=0.4)
plt.scatter(X_proj[:, 0], X_proj[:, 1], alpha=0.6)
```

Out[ ]:
```
<matplotlib.collections.PathCollection at 0x7fbf21874310>
```



In [ ]:
```
np.corrcoef(X.T)
```

Out[ ]:
```
array([[1.        , 0.98960585],
       [0.98960585, 1.        ]])
```

In [ ]:
```
np.corrcoef(X_proj.T)
```

Out[ ]:
```
array([[1., 1.],
       [1., 1.]])
```

Let's verify some othe properties we saw from class. First let's compute the eigenvalue decomposition of $X^T X$

In [ ]:
```
lmbda, v = np.linalg.eig(np.dot(X.T,X))
print(lmbda)
print(v)
```

```
[  139.2471242  30023.15310264]
[[-0.81762394 -0.57575263]
 [ 0.57575263 -0.81762394]]
```

In [ ]:
```
# top PC
print(pca.components_.T)
```

```
[[-0.57575738]
 [-0.8176206 ]]
```

Let's recompute it to see both PCs.

In [ ]:
```
pca = PCA(n_components=2)
pca.fit(X)
print(pca.components_.T)
```

```
[[-0.57575738 -0.8176206 ]
 [-0.8176206   0.57575738]]
```

In [ ]:
```
print("Singular values")
sigma = pca.singular_values_
print( round(sigma[0],1) )
print( round(sigma[1],1) )

print("Square root of eigevalues of X'X")
print( round( np.sqrt(lmbda[0]),1) )
print( round( np.sqrt(lmbda[1]),1) )
```

```
Singular values
173.3
11.8

Square root of eigevalues of X'X
11.8
173.3
```

In [ ]:
```
lmbda2, v2 = np.linalg.eig(np.dot(X,X.T))
print(lmbda2)
```

```
[3.00231531e+04+0.00000000e+00j 1.39247124e+02+0.00000000e+00j
 2.32950490e-12+2.77257117e-12j ... 2.33358686e-18+1.98694423e-18j
 2.33358686e-18-1.98694423e-18j 4.73625770e-19+0.00000000e+00j]
```

The complex numbers here are due to numerical issues. Same thing for the values that are close to 0. We can resolve this easily as follows:

In [ ]:
```
print(lmbda2.real)
```

```
[3.00231531e+04 1.39247124e+02 2.32950490e-12 ... 2.33358686e-18
 2.33358686e-18 4.73625770e-19]
```

Again, only the first two eigenvalues are truly non-zero, the remaining ones appear to be non-zero due to numerical errors. This can be verified by checking the rank which is

equal to 2 as expected (**why?**)

In [ ]:
```
np.linalg.matrix_rank(np.dot(X,X.T))
```

Out[ ]: 2

Finally let's look the singular value decomposition of X. What do you observe?

In [ ]:
```
np.linalg.svd(X)
```

Out[ ]:
```
(array([[-1.46198617e-02,  3.88395863e-04,  6.84675187e-04, ...,
         -6.98721727e-03, -2.30137272e-02,  3.17085758e-02],
        [ 1.10916467e-02, -1.19653388e-02,  1.16351034e-02, ...,
         -1.23752897e-02, -1.37666116e-03,  1.50918479e-02],
        [-5.31675496e-04,  1.16436239e-02,  9.99865804e-01, ...,
          1.47084625e-04,  3.18209254e-05, -1.95491909e-04],
        ...,
        [-5.65912661e-03, -1.30365229e-02,  1.47658222e-04, ...,
          9.99799705e-01, -1.75912402e-04,  4.04464839e-04],
        [-2.27439157e-02, -3.76720121e-03,  3.39225631e-05, ...,
         -1.78079871e-04,  9.99475706e-01,  7.43233408e-04],
        [ 2.99616573e-02,  1.83143177e-02, -1.98316410e-04, ...,
          4.06725017e-04,  7.40841576e-04,  9.98778644e-01]]),
 array([173.27190512,  11.80030187]),
 array([[ 0.57575263,  0.81762394],
        [ 0.81762394, -0.57575263]]))
```

Let's see this in greater detail with another example. Specifically, how would you compute the PCA assuming you could just call an SVD function?

In [ ]:
```
def compare_pca(data):
    # Perform PCA using SVD
    U, s, VT = np.linalg.svd(data - np.mean(data, axis=0))
    pcs_svd = VT.T

    # Perform PCA using scikit-learn
    pca = PCA()
    pca.fit(data)
    pcs_sklearn = pca.components_.T

    return pcs_svd, pcs_sklearn
```

In [ ]:
```
data = np.random.rand(100, 5)
pcs_svd, pcs_sklearn = compare_pca(data)
print(pcs_svd)
print(pcs_sklearn)
```

```
[[ 0.339028    0.1796527  -0.54724711 -0.32484585 -0.66916418]
 [-0.44640461  0.18064378  0.03743157 -0.85127919  0.20497163]
 [ 0.51147024 -0.09097053 -0.477831   -0.14139456  0.69412367]
 [-0.08931328 -0.96223006 -0.07160338 -0.19652702 -0.14962128]
 [ 0.64513895 -0.03050238  0.68240015 -0.3334421  -0.07753538]]
[[ 0.339028    0.1796527  -0.54724711 -0.32484585 -0.66916418]
 [-0.44640461  0.18064378  0.03743157 -0.85127919  0.20497163]
 [ 0.51147024 -0.09097053 -0.477831   -0.14139456  0.69412367]
 [-0.08931328 -0.96223006 -0.07160338 -0.19652702 -0.14962128]
 [ 0.64513895 -0.03050238  0.68240015 -0.3334421  -0.07753538]]
```

In [ ]:

```
pcs_svd-pcs_sklearn
```

Out[ ]:

```
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])
```

SVD as the best k-rank approximation

SVD is commonly used for image compression, reducing the dimensionality of a dataset, and for data compression in general. By retaining only the largest singular values and their corresponding singular vectors, we can reduce the size of a dataset while preserving most of the information (**question**: what is the space we use when we perform a k-rank approximation?)

By retaining only the largest singular values and their corresponding singular vectors, we can effectively reduce the size of a dataset while still preserving most of its important information. This is because the largest singular values and their vectors capture the most variation in the original data, while the smaller singular values and vectors capture noise and other unimportant information.

In the context of image compression, for example, we can apply SVD to the pixel values of an image matrix to obtain the left singular vector matrix, singular value diagonal matrix, and right singular vector matrix. We can then choose to retain only the top k singular values and their corresponding vectors, effectively compressing the image into a smaller set of values. This compressed representation can then be used to reconstruct a close approximation of the original image, with the degree of approximation depending on the number of singular values retained.

The Eckart-Young-Mirsky theorem guarantees that the truncated SVD X_k minimizes the Frobenius norm approximation error among all rank-k matrices, i.e., all matrices M of rank at most k. In other words, for any rank-k matrix M, we have:

$$||X - X_k||_F \leq ||X - M||_F$$

This means that the truncated SVD approximation is the best rank-k approximation of X in terms of the Frobenius norm.

In [ ]:
```
# Load the image
img = cv2.imread('larnaca.jpeg', cv2.IMREAD_GRAYSCALE)

# Apply SVD to the image
U, S, VT = np.linalg.svd(img)

# Define the number of singular values to retain (k)
k = 5

# Reconstruct the image using only k singular values
img_reconstructed = U[:, :k] @ np.diag(S[:k]) @ VT[:k, :]

k2 = 50
img_reconstructed2 = U[:, :k2] @ np.diag(S[:k2]) @ VT[:k2, :]

fig, axs = plt.subplots(1, 3, figsize=(40, 32))

# Plot the original and reconstructed images
plt.subplot(1, 3, 1)

plt.imshow(img, cmap='gray')
plt.title('Original Image')

plt.subplot(1, 3, 2)
plt.imshow(img_reconstructed, cmap='gray')
plt.title('Reconstructed Image (k = %d)' % k)


plt.subplot(1, 3, 3)
plt.imshow(img_reconstructed2, cmap='gray')
plt.title('Reconstructed Image (k = %d)' % k2)

plt.subplots_adjust(wspace=0.7)

plt.show()
```

In [ ]: