CS 236756 - Technion - Intro to Machine Learning

Tal Daniel

Tutorial 03 - Linear Algebra & SVD

Agenda

- [Linear Algebra Refresher](#)
- [Eigen Values and Vectors Decomposition](#)
- [Singular Value Decomposition (SVD)](#))
- [Recommended Videos](#)
- [Credits](#)

Useful Resource

[The Matrix Cookbook (http://www2.imm.dtu.dk/pubdb/views/edoc_download.php/3274/pdf/imm3274.pdf)](http://www2.imm.dtu.dk/pubdb/views/edoc_download.php/3274/pdf/imm3274.pdf)

In [1]:
```python
# imports for the tutorial
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib notebook
```

Linear Algebra Refresher

Vectors

- Geometric object that has both a magnitude and direction
    - $x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = (x_1, x_2, \dots, x_n)^T \in \mathcal{R}^n$
- Magnitude of a vector: $||x|| = \sqrt{x^T x} = \sqrt{x_1^2 + x_2^2 + \dots + x_n^2}$
- **Cardinality** of a vector - the number of non zero elements

In [2]:
```python
# let's see some vectors
v = np.random.randint(low=-20, high=20, size=(6, 1))
print("v:")
print(v)
print("v^T:")
print(v.T)
```

```
v:
[[ 16]
 [  0]
 [ 19]
 [-16]
 [ -9]
 [ 10]]
v^T:
[[ 16   0  19 -16  -9  10]]
```

In [3]:
```python
print("magnitude of v:")
print(np.sqrt(np.sum(np.square(v))))
print("cardinality- non zero elements:")
print(np.sum(v != 0))
```

```
magnitude of v:
32.46536616149585
cardinality- non zero elements:
5
```

Inner Product Space

- A mapping $\langle \cdot, \cdot \rangle : V \times V \rightarrow F$ that satisfies:
  - Conjucate Symmetry: $\langle x, y \rangle = \overline{\langle y, x \rangle}$
  - Linearity in the First Argument:
    - $\langle a \cdot x, y \rangle = a \cdot \langle x, y \rangle$
    - $\langle x + z, y \rangle = \langle x, y \rangle + \langle z, y \rangle$
  - Positive-definiteness:
    - $\langle x, x \rangle \geq 0$
    - $\langle x, x \rangle = 0 \rightarrow x = 0$

- Common Inner Products:
  - Real Vector: $\langle x, y \rangle = x^T y$
  - Real Matrix: $\langle A, B \rangle = trace(AB^T)$
  - Random Variables: $\langle x, y \rangle = \mathbb{E}[x \cdot y]$

- Properties of **Dot Product**:
  - Distributiveness:
    - $(a + b) \cdot c = a \cdot c + b \cdot c$
    - $a \cdot (b + c) = a \cdot b + a \cdot c$
  - Linearity: $(\lambda a) \cdot b = a \cdot (\lambda b) = \lambda (a \cdot b)$
  - Symmetry: $a \cdot b = b \cdot a$
  - Non-Negativity: $\forall a \neq 0, a \cdot a > 0, a \cdot a = 0 \iff a = 0$

In [4]:
```python
# Let's see some dot products
a = np.ones((5,1))
b = np.random.randint(low=-10, high=10, size=(5,1))
print("a:")
print(a)
print("b:")
print(b)
print("a.T.dot(b)=")
print(a.T.dot(b))
print("the same as a.T @ b:")
print(a.T @ b)
```

```
a:
[[1.]
 [1.]
 [1.]
 [1.]
 [1.]]
b:
[[ 3]
 [-4]
 [ 5]
 [ 8]
 [-4]]
a.T.dot(b)=
[[8.]]
the same as a.T @ b:
[[8.]]
```

In [5]:
```python
print("a + 0.5=")
print(a + 0.5)
print("(a + 2 * a).T @ b")
print((a + 2 * a).T @ b)
print("the same as a.T @ b + (2 * a).T @ b")
print(a.T @ b + (2 * a).T @ b)
```

```
a + 0.5=
[[1.5]
 [1.5]
 [1.5]
 [1.5]
 [1.5]]
(a + 2 * a).T @ b
[[24.]]
the same as a.T @ b + (2 * a).T @ b
[[24.]]
```

Outer Product

- Let:
  - $a = (a_1, a_2, \ldots, a_n)^T$
  - $b = (b_1, b_2, \ldots, b_n)^T$
- The outer product ab^T:

$$ab^T = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} [b_1, b_2, \ldots, b_n] = \begin{pmatrix} a_1b_1 & a_1b_2 & \cdots & a_1b_n \\ a_2b_1 & a_2b_2 & \cdots & a_2b_n \\ \vdots & \vdots & \ddots & \vdots \\ a_nb_1 & a_nb_2 & \cdots & a_nb_n \end{pmatrix}$$

In [6]:
```python
# outer product
a = np.random.random(size=(5,1))
print("a:")
print(a)
b = np.random.random(size=(5,1))
print("b:")
print(b)
```

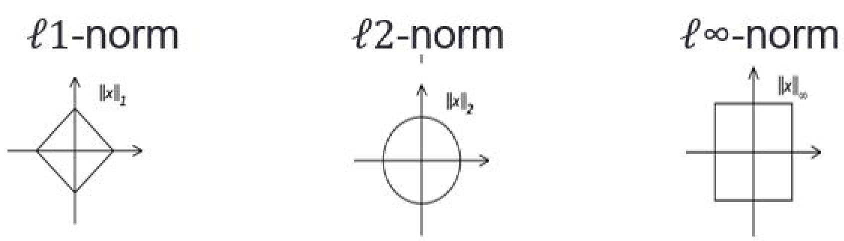
```
a:
[[0.68496376]
 [0.51514789]
 [0.97263803]
 [0.47948046]
 [0.97063678]]
b:
[[0.16180323]
 [0.64818973]
 [0.00683339]
 [0.5219497 ]
 [0.02569252]]
```

In [7]:
```python
ab_t = a @ b.T
print("outer product: a @ b.T = ")
print(ab_t)
```

```
outer product: a @ b.T =
[[0.11082935 0.44398648 0.00468062 0.35751663 0.01759844]
 [0.08335259 0.33391357 0.00352021 0.26888128 0.01323545]
 [0.15737597 0.63045398 0.00664641 0.50766812 0.02498952]
 [0.07758149 0.31079431 0.00327648 0.25026468 0.01231906]
 [0.15705217 0.62915679 0.00663274 0.50662357 0.0249381 ]]
```

Vector Norms

- A norm on a vector sapce Ω is a function $f : \Omega \rightarrow \mathcal{R}$ with the following properties:
  - Positive Scalability: $f(ax) = |a|f(x)$
  - Triangle Inequality: $f(x+y) \leq f(x) + f(y)$
  - If $f(x) = 0 \rightarrow x = 0$
- l_1 norm: $||x||_1 = \sum_{i=1}^{n} |x_i|$
- l_2 norm: $||x||_2 = \sqrt{\sum_{i=1}^{n} |x_i|^2}$
  - For **Vectors**: $||x||_2^2 = x^T x$
  - l_2-distance: $||x-y||_2^2 = (x-y)^T(x-y) = ||x||_2^2 - 2x^Ty + ||y||_2^2$
- l_p norm: $||x||_p = (\sum_{i=1}^{n} |x_i|^p)^{\frac{1}{p}}$
- l_∞ norm: $||x||_\infty = \max(|x_1|, |x_2|, \ldots, |x_n|)$



```
In [8]: # norms and distance
        a = np.random.random(size=(5,1))
        print("a:")
        print(a)
        print("l-1 norm: ")
        print(np.sum(abs(a)))
        print("l-2 norm: ")
        print(np.sqrt(np.sum(np.square(a))))
        print("l-infinity norm:")
        print(np.max(abs(a)))
```

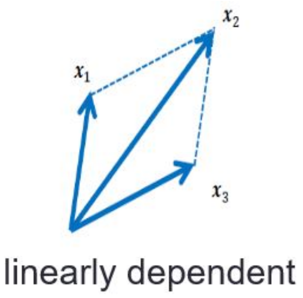
```
a:
[[0.20110422]
 [0.3103417 ]
 [0.25755954]
 [0.84291866]
 [0.00855558]]
l-1 norm:
1.6204796988041368
l-2 norm:
0.9558644554276373
l-infinity norm:
0.8429186563888088
```

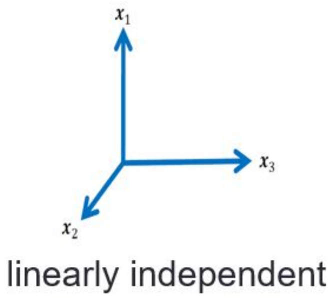
```
In [9]: b = np.random.random(size=(5,1))
        print("b:")
        print(b)
        print("l-2 distance between a and b:")
        print(np.sqrt((a - b).T @ (a - b)))
```

```
b:
[[0.59011591]
 [0.77681828]
 [0.31464032]
 [0.78600795]
 [0.85952156]]
l-2 distance between a and b:
[[1.04860414]]
```

Linear Dependency

- Given a set of vectors $X = \{x_1, x_2, \dots, x_n\}$, a **linear combination** of vectors is written as:
$$ax = a_1x_1 + a_2x_2 + \dots + a_nx_n$$
- $x_i \in X$ is **linearly dependent** if it can be written as linear combination of $X \setminus \{x_i\}$



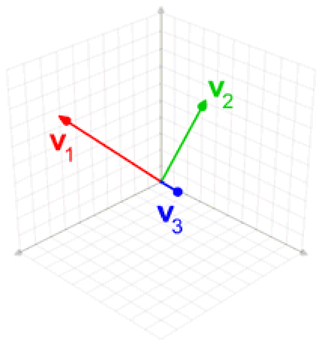


Basis

- A basis is a **linearly independent** set of vectors that spans the "whole sapce"
- Every vector in the space can be written as a linear combination of vectors in the basis
  - For example, **the standard basis (unit vectors)**: $\{e_i \in \mathcal{R}^n | e_i = (0, 0, \dots, 0, 1, 0, \dots, 0)^T\}$
    - $x^T = (3, 2, 5)^T = 3(1, 0, 0)^T + 2(0, 1, 0)^T + 5(0, 0, 1)^T = 3e_1^T + 2e_2^T + 5e_3^T$

- **Projection** of a vector: $x \cdot e_i = x^T e_i = e_i^T x$
- The basis vectors suffice:
  - Orthogonal - $e_i^T e_j = 0$
  - Normalized - $e_i^T e_i = 1$
  - Orthogonal + Normalized = Orthonormal
  - If A is **orthogonal** then:
    - A is a square matrix
    - The columns of A are **orthonormal** vectors
    - $A^T A = AA^T = I \rightarrow A^T = A^{-1}$

- **Change of Basis** - suppose that we have a basis not necessarily orthonormal $B = \{b_1, b_2, \ldots, b_n\}, b_i \in \mathcal{R}^m$
  - Vector in the **new** basis is represented with a **matrix-vector** multiplication
  - The Identity matrix I maps a vector to itself
  - Basis change can be decomposed to: **rotation** matrix and **scale** matrix
  - Using an **orthonormal** basis means only a **rotation** around the origin
  - **Gram-Schmidt Orthonormaliztion Process**: [Link (https://en.wikipedia.org/wiki/Gram%E2%80%93Schmidt_process)](https://en.wikipedia.org/wiki/Gram%E2%80%93Schmidt_process)



By [Lucas V. Barbosa (//commons.wikimedia.org/wiki/User:Kieff)](//commons.wikimedia.org/wiki/User:Kieff) - Own work, Public Domain, [Link (https://commons.wikimedia.org/w/index.php?curid=24396471)](https://commons.wikimedia.org/w/index.php?curid=24396471)

In [6]:

```python
# Gram-Schmidt Algorithm
def gram_schmidt(V):
    """
    Implements Gram-Schmidt Orthonormaliztion Process.
    Parameters:
        V - matrix such that each column is a vector in the original basis
    Returns:
        U - matrix with orthonormal vectors as columns
    """
    n, k = np.array(V, dtype=np.float).shape  # get dimensions
    # initialize U matrix
    U = np.zeros_like(V, dtype=np.float)
    U[:,0] = V[:,0] / np.sqrt(V[:,0].T @ V[:,0])
    for i in range(1, k):
        U[:,i] = V[:,i]
        for j in range(i - 1):
            U[:,i] = U[:,i] - ((U[:,i].T @ U[:,j]) / (U[:,j].T @ U[:,j])) * U[:, j]
        # normalize
        U[:,i] = U[:,i] / np.sqrt(U[:,i].T @ U[:,i])
    return U

v1 = [3.0, 1.0]
v2 = [2.0, 2.0]
v = np.stack((v1, v2), axis=1)
print("V:")
print(v)
U = gram_schmidt(v)
print("U:")
print(U)
```

```
V:
[[3. 2.]
 [1. 2.]]
U:
[[0.9486833  0.70710678]
 [0.31622777 0.70710678]]
```

Matrix Operations

- Addition
  - Commutative: $A+B=B+A$
  - Associative: $(A+B)+C=A+(B+C)$
- Multiplication - **PAY ATTENTION TO DIMENSTIONS**
  - Associative: $A(BC)=(AB)C$
  - Distributive: $A(B+C)=AB+AC$
  - Non-comutative (**!**): $AB \neq BA$

- Transpose
  - $(A^T)_{ij}$
  - $(A^T)^T = A$
  - $(AB)^T = B^T A^T$
- Inverse - **MAKE SURE CONDITIONS APPLY**
  - **Positive Semi-definite (PSD)** - Matrix M is called *PSD* if for every non-zero column vector z, the scalar $z^T M z \geq 0$
  - **Every positive definite matrix is invertible** and its inverse is also positive definite
  - $(A^{-1})^{-1} = A$
  - $(AB)^{-1} = B^{-1}A^{-1}$
  - $(A^T)^{-1} = A^{-T}$
  - Inverse of 2x2 matrix: see tutorial 1

In [10]:
```python
# inverse
A = np.random.rand(5, 5)
print("A:")
print(A)
print("inverse of A:")
print(np.linalg.inv(A))
```

```
A:
[[0.56274722 0.57692677 0.31759767 0.9135175  0.39388189]
 [0.3260898  0.73720574 0.3526661  0.02961814 0.16645483]
 [0.01740472 0.24892669 0.4684225  0.60255541 0.11491183]
 [0.60243149 0.97287256 0.72073364 0.33608398 0.94720029]
 [0.3300669  0.15559865 0.27349031 0.41204091 0.83342534]]
inverse of A:
[[  21.57251296 -108.00106195  -17.70755954   87.22168674  -85.31216784]
 [ -14.53515995   78.79387459   11.54867247  -62.74719293   60.8531958 ]
 [  14.80752023  -84.10430348  -11.05067623   68.39955569  -66.41395368]
 [  -4.51707378   27.97302751    4.82828719  -23.33701698   22.40506618]
 [  -8.45572468   41.8310235     6.09595381  -33.73598262   34.3423877 ]]
```

Matrix Rank

- The rank of a matrix is the **maximal number of linearly independent** columns or rows of a matrix
- $A \in \mathcal{R}^{m \times n} \rightarrow rank(A) \leq \min(m,n)$
- $rank(A) = rank(A^T)$
- $rank(A^TA) = rank(A)$
- $rank(A+B) \leq rank(A) + rank(B)$
- $rank(AB) \leq \min(rank(A), rank(B))$
- A is **full rank** if $rank(A) = \min(m,n)$
- **Singular Matrix** - has dependent rows (and at least one zero eigen-value)

In [11]:
```python
A = np.random.randint(low=0, high=4, size=(5,5))
print("A:")
print(A)
print("rank(A):")
print(np.linalg.matrix_rank(A))
```

```
A:
[[0 3 3 3 1]
 [1 1 1 3 3]
 [1 1 2 2 0]
 [2 0 3 1 2]
 [3 1 2 1 1]]
rank(A):
5
```

Range & Nullspace

- **Range** (of a matrix) - the span of the columns of the matrix, denoted by the set:
$$\mathcal{R}(A) = \{y | y = Ax\}$$
- **Nullspace** (of a matrix) - the set of vectors that when multiplied by the matrix result in 0, given by the set:
$$\mathcal{N}(A) = \{x | Ax = 0\}$$

Determinant

Let $A = \begin{pmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{pmatrix}$, a **square matrix**, then:

$$det(A) = |A| = \begin{vmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{vmatrix} = x_1 \begin{vmatrix} y_1 & z_2 \\ y_3 & z_3 \end{vmatrix} - x_2 \begin{vmatrix} y_1 & z_1 \\ y_3 & z_3 \end{vmatrix} + x_3 \begin{vmatrix} y_1 & z_1 \\ y_2 & z_2 \end{vmatrix}$$

$$= x_1(y_2z_3 - z_2y_3) - x_2(y_1z_3 - z_1y_3) + x_3(y_1z_2 - z_1y_2)$$

- $det(A) = 0 \iff A$ is **singular** (at least one eigen-value is zero)
- If A is diagonal, then $det(A)$ is the prodcut of the diagonal elements (the eigen-values)
- $det(AB) = det(A)det(B)$
- $det(A^{-1}) = det(A)^{-1}$
- $det(\lambda A) = \lambda^n det(A)$

In [12]:
```
# determinant
A = np.random.randn(5,5)
print("A:")
print(A)
print("det(A):")
print(np.linalg.det(A))
```

```
A:
[[-0.11682683 -0.60007878  0.20168493 -0.41938087 -1.44710738]
 [-0.77820688  0.97102027 -0.95386608 -0.81321839  0.83334389]
 [-1.44149225 -0.44278972 -0.07846115  0.59192462  0.21563895]
 [-0.75701366 -1.49163516 -0.2865721  -0.46047925 -0.01296227]
 [ 1.250518    1.20554034 -0.14421321  0.44739448 -0.14740781]]
det(A):
3.073911389887483
```

Solve Linear Equation Analytically

- Definitions:
  - $A \in \mathcal{R}^{n \times n}$
  - $x, b \in \mathcal{R}^{n \times 1}$
- The problem: find the solution of $Ax = b$
- Solution: if A is PSD (and thus invertible), then $x = A^{-1}b$
- What if $A \in \mathcal{R}^{m \times n}, x \in \mathcal{R}^{n \times 1}, b \in \mathcal{R}^{m \times 1}$?
  - A is no longer invertible!
- The problem redefined: find x that minimzes the distance from Ax to b, or more formally:

  $$\underset{x}{\operatorname{argmin}} ||Ax - b||_2^2$$

  (also called **least-squares** solution)

Reminder (Tutorial 01) - Vector & Matrix Derivatives

- $\nabla_x Ax = A^T$
- $\nabla_x x^T Ax = (A + A^T)x$
- $\frac{\partial}{\partial A} \ln |A| = A^{-T}$
- $\frac{\partial}{\partial A} Tr[AB] = B^T$

Exercise 1 - Least-Squares Solution

Given $A \in \mathcal{R}^{m \times n}$, $x \in \mathcal{R}^{n \times 1}$, $b \in \mathcal{R}^{m \times 1}$

Find x that minimizes the distance from Ax to b, or more formally:

$$\underset{x}{\text{argmin}} ||Ax - b||_2^2$$

Solution 1

$$||Ax - b||_2^2 = (Ax - b)^T (Ax - b) = x^T A^T Ax - x^T A^T b - b^T Ax + b^T b$$
$$\frac{\partial ||Ax - b||_2^2}{\partial x} = 2A^T Ax - 2A^T b = 0 \rightarrow x = (A^T A)^{-1} A^T b$$

In [13]:

```python
# Least Squares Solution
m = 5
n = 4
A = np.random.randint(low=-5, high=10, size=(m,n))
b = np.random.randint(low=-10, high=3, size=(m,1))
print("A:")
print(A)
print("b:")
print(b)
```

```
A:
[[ 3  2  8  9]
 [-3 -5 -5  2]
 [ 0  5  7  5]
 [ 1 -3  6 -5]
 [ 1  1  8  6]]
b:
[[-2]
 [-7]
 [-3]
 [-3]
 [ 0]]
```

In [14]:

```python
print("Least Squares solution for x:")
x = np.linalg.inv(A.T @ A) @ A.T @ b
print(x)
```

```
Least Squares solution for x:
[[ 1.54495052]
 [ 0.65381817]
 [-0.47872248]
 [-0.27042109]]
```

Solve Linear Equation Non-Analytically

Eigenvalues and Eigenvectors

- Definition: Matrix A with **Eigenvalue** $\lambda \in \mathbb{C}$ and **Eigenvector** $x \in \mathbb{C}^n$ if
$$Ax = \lambda x, x \neq 0$$
- Finding eigenvalues and eigenvectors
  - Find eigenvalues by finding the roots of the polynomial generated by:
$$det(\lambda I - A) = |\lambda I - A| = 0$$
  - For each eigenvalue λ, find its corresponding eigenvector x by solving:
$$Ax = \lambda x$$

- Example: $M = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix} \rightarrow |\lambda I - M| = \begin{vmatrix} 2-\lambda & 1 \\ 1 & 2-\lambda \end{vmatrix} = 3 - 4\lambda + \lambda^2 \rightarrow \lambda_{1,2} = 1, 3 \rightarrow x_{\lambda=1} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, x_{\lambda=3} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$
- Eigenvalues Properties
  - $det(\Lambda) = |\Lambda| = \prod_{i=1}^{n} \lambda_i$
  - $rank(A) = \sum_{i=1}^{n} 1_{\lambda_i \neq 0}$
  - Eigenvalues of a **diagonal** matrix are the diagonal entries
  - A (square) matrix is said to be **diagonalizable** if it can be rewritten as: $A = X\Lambda X^{-1}$
- Eigenvalues of **Symmetric Matrices**:
  - Eigenvalues are **real**
  - Eigenvectors of **real symmetric** matrices are orthonormal

In [15]:

```
# eigenvalues and eigenvectors
A = np.random.randint(low=-10, high=10, size=(5,5))
eig, vec = np.linalg.eig(A)
print("A:")
print(A)
```

```
A:
[[-4 -9  7  8  1]
 [ 6 -8 -5 -3 -9]
 [ 0  9 -6  0  3]
 [ 8  2 -6  0 -6]
 [ 3  3  2  4 -1]]
```

In [16]:

```
print("eigenvalues:")
print(eig)
print("eigenvectors:")
print(vec)
```

```
eigenvalues:
[-9.29854727+11.14091902j -9.29854727-11.14091902j
  3.93378061 +0.j         -3.01573245 +0.j
 -1.32095361 +0.j        ]
eigenvectors:
[[ 0.2627824 -0.45749602j  0.2627824 +0.45749602j -0.68051908+0.j
   0.33207203+0.j          0.54461743+0.j        ]
 [-0.57285962+0.j         -0.57285962-0.j          0.20089578+0.j
  -0.39152492+0.j         -0.32875482+0.j        ]
 [ 0.13729842+0.38931395j  0.13729842-0.38931395j  0.00207705+0.j
  -0.45532511+0.j         -0.15673315+0.j        ]
 [-0.31676371+0.31808551j -0.31676371-0.31808551j -0.3762196 +0.j
  -0.09140478+0.j         -0.14305208+0.j        ]
 [ 0.12184535+0.08182002j  0.12184535-0.08182002j -0.59580967+0.j
   0.72163745+0.j          0.74181059+0.j        ]]
```

Eigen Decomposition

- **Eigen-decomposition** (also **spectral decomposition**) - factorization of a matrix into a canonical form, that is, the matrix is represented in terms of its **eigenvalues and eigenvectors**.
- **Only** diagonalizable matrices can be factorized
- Formally:
  - Denote Λ as a matrix with eigenvalues on the diagonal
  - Denote Q as a matrix where the columns are the eigenvectors
  - Let A be a square $n \times n$ matrix with N linearly **independent** columns. Then A can factorized as:
$$A = Q\Lambda Q^{-1}$$

What If A Is Non-Square?

Singular Value Decomposition (SVD)

- In linear algebra, the singular-value decomposition (SVD) is a factorization of a real or complex matrix. It is the generalization of the eigendecomposition of a positive semidefinite normal matrix (for example, a symmetric matrix with positive eigenvalues) to any $m \times n$ matrix via an extension of the polar decomposition.
- Definition:

$$A_{[m \times n]} = U_{[m \times r]} \Sigma_{[r \times r]} (V_{[n \times r]})^T$$

- A - Input Data matrix
  - $m \times n$ matrix (e.g. m documents and n terms that can appear in each document)
- U - Left Singular vectors
  - $m \times r$ matrix (e.g. m documents and r concepts)
  - $U = eig(AA^T)$

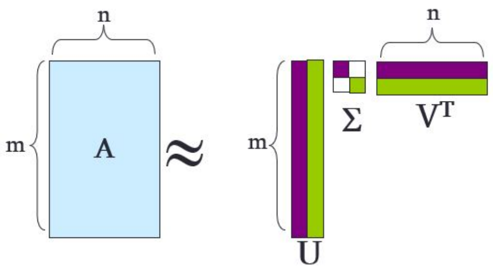
- Σ - Singular values
  - $r \times r$ **diagonal** matrix (strength of each 'concept')
  - r represnts the **rank** of matrix A
  - $\Sigma = diag\left(\sqrt{eigenvalues(A^TA)}\right)$
  - **Singular Values** definition: the singular values of a matrix $X \in \mathbb{R}^{M \times N}$ are the *square root* of the **eigenvalues** of the matrix $X^TX \in \mathbb{R}^{N \times N}$. If $X \in \mathbb{R}^{N \times N}$ already, then the singular values are the eigenvalues.
- V - Right Singular vectors
  - $n \times r$ matrix (e.g. n terms and r concepts)
  - $V = eig(A^TA)$

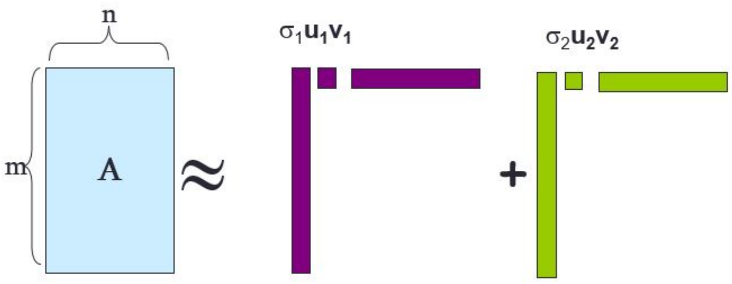
- Illustration:

  First, we see the unit disc in blue together with the two canonical unit vectors. We then see the action of M, which distorts the disk to an ellipse. The SVD decomposes M into three simple transformations: an initial rotation V^*, a scaling Σ along the coordinate axes, and a final rotation U. The lengths σ_1 and σ_2 of the semi-axes of the ellipse are the singular values of M, namely $\Sigma_{1,1}$ and $\Sigma_{2,2}$.
- By [Kieff (//commons.wikimedia.org/wiki/User:Kieff)](//commons.wikimedia.org/wiki/User:Kieff) - Own work, Public Domain, [Link (https://commons.wikimedia.org/w/index.php?curid=11416486)](https://commons.wikimedia.org/w/index.php?curid=11416486)

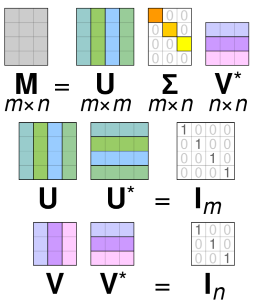
- Another way to look at SVD:

$$A \approx U\Sigma V^T = \sum_i \sigma_i u_i \circ v_i^T$$





- **SVD Properties**
  - It is **always** possible to decompose a **real** matrix A to $A = U\Sigma V^T$ where
    - U, Σ, V are **uniuqe**
    - U, V are column **orthonormal**
      - $U^T U = I, V^T V = I$
    - Σ is **diagonal**
      - Entries (the singular values) are positive and **sorted** in decreasing order ($\sigma_1 \geq \sigma_2 \geq \ldots \geq 0$)
  - Proof of uniqueness (http://www.mpi-inf.mpg.de/~bast/ir-seminar-ws04/lecture2.pdf[image.png](attachment:image.png))



- [Image Source (https://en.wikipedia.org/wiki/Singular_value_decomposition)](https://en.wikipedia.org/wiki/Singular_value_decomposition)

SVD Example - Users-to-Movies

We are given a dataset of user's rating (1 to 5) for several movies of 3 genres (concepts) and we wish to use SVD to decompose to the following components:

- User-to-Concept - which genres the users prefer: U matrix
- Concepts - what is the strength of each genre in the dataset: Σ - strength of each concept (the singular values)
- Movie-to-Concept - for each movie, what genres are the most dominant: V matrix

In [17]:
```
# load the dataset and create a pandas DataFrame
u_t_m = np.array([[1,1,1,0,0], [3,3,3,0,0], [4,4,4,0,0], [5,5,5,0,0], [0,2,0,4,4], [0,0,0,5,5], [0,1,0,2,2
]])
print("User-to-Movies matrix:")
# print(u_t_m)
u_t_m_df = pd.DataFrame(u_t_m, columns=['Matrix', 'Alien', 'Serenity', 'Casablanca', 'Amelie'],
                        index=['User 1', 'User 2','User 3', 'User 4', 'User 5', 'User 6', 'User 7'])
u_t_m_df
```

```
User-to-Movies matrix:
```

Out[17]:

	Matrix	Alien	Serenity	Casablanca	Amelie
User 1	1	1	1	0	0
User 2	3	3	3	0	0
User 3	4	4	4	0	0
User 4	5	5	5	0	0
User 5	0	2	0	4	4
User 6	0	0	0	5	5
User 7	0	1	0	2	2

In [18]:
```
# perform SVD for 3 concepts
u, s, vh = np.linalg.svd(u_t_m, full_matrices=False)
```

In [19]:
```
print("U of size", u[:,:3].shape, ":")
print(u[:,:3].astype(np.float16))
```

```
U of size (7, 3) :
[[-0.1376   0.0236   0.01081]
 [-0.4128   0.07086  0.03244]
 [-0.5503   0.0944   0.04324]
 [-0.688    0.11804  0.05405]
 [-0.1528  -0.5913  -0.654  ]
 [-0.0722  -0.7314   0.678  ]
 [-0.0764  -0.2957  -0.327  ]]
```

In [21]:
```
print("Singular values:")
print("as a matrix:")
print(np.diag(s[:3]).astype(np.float16))
```

```
Singular values:
as a matrix:
[[12.484  0.     0.   ]
 [ 0.     9.51   0.   ]
 [ 0.     0.     1.346]]
```

In [22]:
```
print("V of size", vh[:3,:].shape, ":")
print(vh[:3,:].astype(np.float16))
```

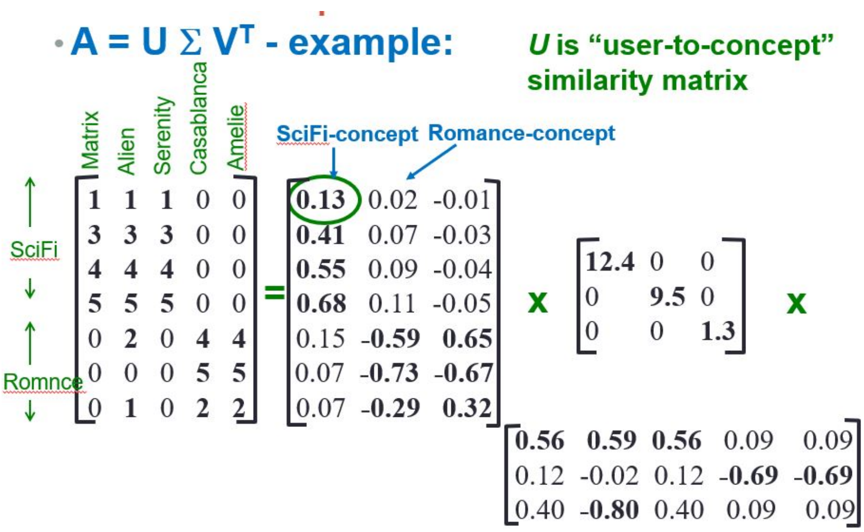
```
V of size (3, 5) :
[[-0.5625  -0.593   -0.5625  -0.09015 -0.09015]
 [ 0.1266  -0.02878  0.1266  -0.6953  -0.6953 ]
 [ 0.4097  -0.8047   0.4097   0.09125  0.09125]]
```

In [23]:
```python
# reconstruct the user-to-movie matrix
A_aprox = u[:,:3] @ np.diag(s[:3]) @ vh[:3,:]
A_aprox_df = pd.DataFrame(A_aprox.astype(np.float16), columns=['Matrix', 'Alien', 'Serenity', 'Casablanca'
, 'Amelie'],
                          index=['User 1', 'User 2','User 3', 'User 4', 'User 5', 'User 6', 'User 7'])
print("reconstruction of user-to-movie:")
A_aprox_df
```

```
reconstruction of user-to-movie:
```

Out[23]:

	Matrix	Alien	Serenity	Casablanca	Amelie
User 1	1.0	1.0	1.0	0.0	0.0
User 2	3.0	3.0	3.0	-0.0	-0.0
User 3	4.0	4.0	4.0	0.0	-0.0
User 4	5.0	5.0	5.0	-0.0	-0.0
User 5	0.0	2.0	-0.0	4.0	4.0
User 6	0.0	0.0	-0.0	5.0	5.0
User 7	0.0	1.0	-0.0	2.0	2.0



Recommended Videos

Warning!

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

Video By Subject

- Basic Linear Algebra - [Mathematics for Machine Learning full Course II Linear Algebra II Part-1 (https://www.youtube.com/watch?v=T3TpdPmTLso)](https://www.youtube.com/watch?v=T3TpdPmTLso)
- SVD - [Lecture 47 — Singular Value Decomposition I Stanford University (https://www.youtube.com/watch?v=P5mlg91as1c)](https://www.youtube.com/watch?v=P5mlg91as1c)

Credits

- Inspired by slides by Elad Osherov and slides from [MMDS (http://www.mmds.org/)](http://www.mmds.org/)
- Icons from [Icon8.com (https://icons8.com/)](https://icons8.com/) - [https://icons8.com (https://icons8.com)](https://icons8.com)
- Datasets from [Kaggle (https://www.kaggle.com/)](https://www.kaggle.com/) - [https://www.kaggle.com/ (https://www.kaggle.com/)](https://www.kaggle.com/)