THE GEORGE WASHINGTON UNIVERSITY
WASHINGTON, DC

SEAS 8510
Analytical Methods for Machine Learning

Lecture 3

Zachary Dennis, Ph.D.

Agenda

9:00 – 9:30		Discussion Groups (30 min)
9:30 – 9:50		Homework 2 Review (20 min)
9:50 – 10:40		Systems of Linear Equations and Matrix Operations (50 min)
10:40 – 10:50		*BREAK (10 min)*
10:50 – 11:45		Matrix Forms and Vector Spaces (55 min)
11:45 – 12:00		Homework 3/Discussion 3 Overview (15 min)

Assignments

Last week: Discussion 2/Homework 2

This week: Discussion 3/Homework 3 – Due on 4/13 at 9:00 AM

Discussion 2

Prompt:

1. Describe a real world example of a system of linear equations.

2. Explain the concept of a Vector Subspace.

Instructions:

- Choose the breakout room associated with your discussion group number (1-6)
- In small group, discuss your responses (15 min)
- Choose a spokesman to summarize your group's responses (trends, stand out comments, areas of similarity or contrast)
- Back in large group, each spokesman take 120 seconds to summarize your group's responses.

Homework 2

Determinants

Use determinant to know if a matrix is invertible

Square matrices only

If det(A) = 0, then A is not invertible, rows of A are linearly dependent, columns are linearly dependent.

If det(A) != 0, then A is invertible, rows of A are linearly independent, columns are linearly independent.

2x2 $$\det\begin{bmatrix} a & b \\ c & d \end{bmatrix} = ad - bc$$

3x3 $$|A| = a \cdot \begin{vmatrix} e & f \\ h & i \end{vmatrix} - b \cdot \begin{vmatrix} d & f \\ g & i \end{vmatrix} + c \cdot \begin{vmatrix} d & e \\ g & h \end{vmatrix}$$

4x4 $$|A| = a \cdot \begin{vmatrix} f & g & h \\ j & k & l \\ n & o & p \end{vmatrix} - b \cdot \begin{vmatrix} e & g & h \\ i & k & l \\ m & o & p \end{vmatrix} + c \cdot \begin{vmatrix} e & f & h \\ i & j & l \\ m & n & p \end{vmatrix} - d \cdot \begin{vmatrix} e & f & g \\ i & j & k \\ m & n & o \end{vmatrix}$$

Pattern is called Laplace expansion.

Eigenvalues/Eigenvectors

Square matrix A.

Eigenvectors have a special relationship with A such that

$$A\vec{x} = \lambda\vec{x}$$

Eigenvalue λ

Eigenvector $\vec{x}$

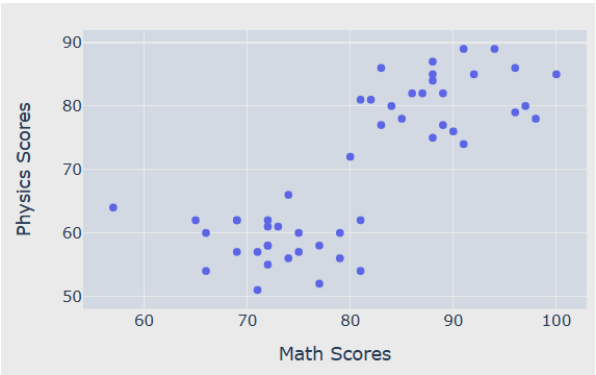
Not the zero vector.

k-Means Clustering (1)

- Unsupervised method of classifying multivariate data into a relatively small number of groups, or categories, based on minimizing distance to the group center
- Algorithm:

1. Initialize *k* centroids as random points in the data space. Each centroid is a class, or category, and the next steps will assign each data observation to each class.
2. Compute the Euclidean distance between each data observation and each centroid
3. Assign each data observation to the group with the closest centroid.
4. Update each centroid as the average of all data observations assigned to that centroid.
5. Repeat steps 2–4 until a convergence criteria is satisfied, or for *N* iterations.

Math Scores: [79. 72. 75. 81. 79. 65. 75. 69. 69. 72. 71. 77. 74. 71.
72. 72. 77. 69. 72. 66. 57. 73. 74. 66. 81. 83. 90. 89.
98. 97. 91. 92. 86. 80. 88. 91. 96. 96. 88. 88. 85. 83.
81. 100. 87. 88. 84. 94. 82. 89.]
Physics Scores: [56. 62. 57. 54. 60. 62. 60. 62. 57. 58. 57. 58. 56. 51. 61. 58. 52. 62.
55. 60. 64. 61. 66. 54. 62. 77. 76. 77. 78. 80. 74. 85. 82. 72. 87. 89.
86. 79. 75. 85. 78. 86. 81. 85. 82. 84. 80. 89. 81. 82.]

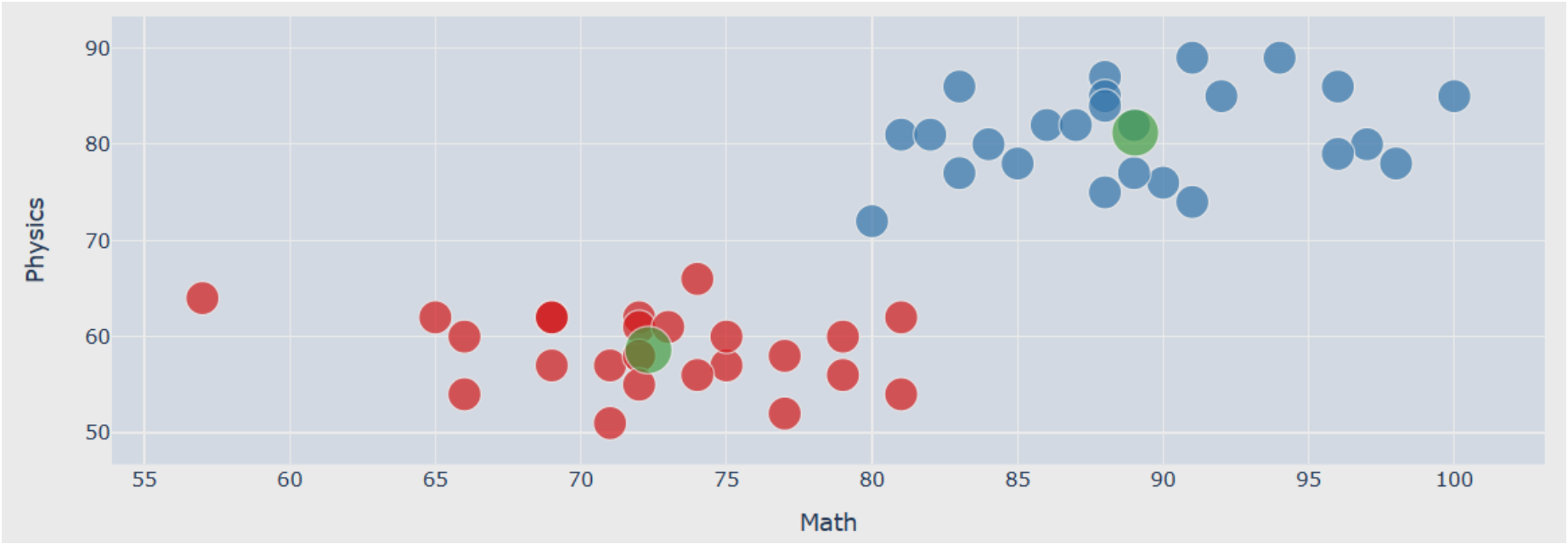


k-Means Clustering (2)

```
def k_means_clustering(data, k, max_iterations=100, tolerance=1e-4):
    # Step 1: Initialize k centroids randomly
    centroids = data[np.random.choice(data.shape[0], k, replace=False)]

    for _ in range(max_iterations):
        # Step 2 & 3: Compute Euclidean distance & Assign data points to the closest centroid
        distances = np.linalg.norm(data - centroids[:, np.newaxis], axis=2)
        assigned_clusters = np.argmin(distances, axis=0)

        # Step 4: Update centroids & Check convergence
        new_centroids = np.array([data[assigned_clusters == idx].mean(axis=0) for idx in range(k)])

        if np.linalg.norm(new_centroids - centroids) < tolerance:
            break

        centroids = new_centroids

    # Adding cluster number as a third column and append centroids
    labeled_data = np.hstack((data, assigned_clusters.reshape(-1, 1) + 1))  # Adding 1 to make cluster numbers
1-indexed
    centroids_with_label = np.hstack((centroids, np.zeros((k, 1))))
    final_data = np.vstack((labeled_data, centroids_with_label))

    return final_data
```

k-Means Clustering (3)

```
# Set the number of clusters
k = 2

# Run k-means clustering and get labeled data
labeled_data_array = k_means_clustering(data, k)

# Print the result
print(labeled_data_array)

labeled_data_array = np.hstack([labeled_data_array, 10+10*(labeled_data_array[:,2]==0).reshape(-1, 1)])

fig = px.scatter(x=labeled_data_array[:, 0], y=labeled_data_array[:, 1], color=labeled_data_array[:, 2].astype(str),
                 size = labeled_data_array[:, 3],
                 title='Clusters and Centroids', labels={'x': 'Math', 'y': 'Physics'},
                 color_discrete_sequence=px.colors.qualitative.Set1)


# Show plot
fig.show()
```



Linear Dependence

- For the following matrix $\mathbf{B} = \begin{bmatrix} 2 & -1 \\ 4 & -2 \end{bmatrix}$
- Notice that for the two columns $\mathbf{b}_1 = [2, 4]^T$ and $\mathbf{b}_2 = [-1, -2]^T$, we can write $\mathbf{b}_1 = -2 \cdot \mathbf{b}_2$
  - This means that the two columns are linearly dependent
- The weighted sum $a_1\mathbf{b}_1 + a_2\mathbf{b}_2$ is referred to as a linear combination of the vectors $\mathbf{b}_1$ and $\mathbf{b}_2$
  - In this case, a linear combination of the two vectors exist for which $\mathbf{b}_1 + 2 \cdot \mathbf{b}_2 = \mathbf{0}$
- A collection of vectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k$ are ***linearly dependent*** if there exist coefficients $a_1, a_2, \ldots, a_k$ not all equal to zero, so that

$$\sum_{i=1}^{k} a_i \mathbf{v_i} = 0$$

- If there is no linear dependence the vectors are ***linearly independent***

Matrix Rank

- For an $n \times m$ matrix, the ***rank*** of the matrix is the largest number of linearly independent columns
- The matrix $\mathbf{B}$ from the previous example has $rank(\mathbf{B}) = 1$, since the two columns are linearly dependent

$$\mathbf{B} = \begin{bmatrix} 2 & 4 \\ -1 & -2 \end{bmatrix}$$

- The matrix $\mathbf{C}$ below has $rank(\mathbf{C}) = 3$, since it has three linearly independent columns
  - I.e., $\mathbf{c}_1 = -1 \cdot \mathbf{c}_4$, $\mathbf{c}_3 = -1 \cdot \mathbf{c}_5$

$$\mathbf{C} = \begin{bmatrix} 1 & 3 & 0 & -1 & 0 \\ -1 & 0 & 1 & 1 & -1 \\ 0 & 3 & 1 & 0 & -1 \\ 2 & 3 & -1 & -2 & 1 \end{bmatrix}$$

Inverse of a Matrix

- For a square $n \times n$ matrix $\mathbf{A}$ with rank n, $\mathbf{A^{-1}}$ is its ***inverse matrix*** if their product is an identity matrix $\mathbf{I}$

$$\mathbf{A}^{-1}\mathbf{A} = \mathbf{A}\mathbf{A}^{-1} = \mathbf{I}$$

- Properties of inverse matrices

$$\left(\mathbf{A}^{-1}\right)^{-1} = \mathbf{A}$$

$$\left(\mathbf{AB}\right)^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}$$

- If $\det(A) = 0$ (i.e., $\text{rank}(A) < n$), then the inverse does not exist
  - A matrix that is not invertible is called a singular matrix
- Note that finding an inverse of a large matrix is computationally expensive
  - In addition, it can lead to numerical instability
- If the inverse of a matrix is equal to its transpose, the matrix is said to be orthogonal matrix

$$\mathbf{A}^{-1} = \mathbf{A}^{T}$$

Inverse of a Matrix

- ***Pseudo-inverse*** of a matrix
  - Also known as Moore-Penrose pseudo-inverse
- For matrices that are not square, the inverse does not exist
  - Therefore, a pseudo-inverse is used
- If $m > n$, then the pseudo-inverse is $\mathbf{A}^{\dagger} = \left(\mathbf{A}^{\mathrm{T}}\mathbf{A}\right)^{-1}\mathbf{A}^{T}$ and $\mathbf{A}^{\dagger}\mathbf{A} = \mathbf{I}$
- If $m < n$, then the pseudo-inverse is $\mathbf{A}^{\dagger} = \mathbf{A}^{T}\left(\mathbf{A}\mathbf{A}^{\mathrm{T}}\right)^{-1}$ and $\mathbf{A}\mathbf{A}^{\dagger} = \mathbf{I}$
  - E.g., for a matrix with dimension $\mathbf{X}_{2\times3}$, a pseudo-inverse can be found of size $\mathbf{X}^{\dagger}_{3\times2}$, so that $\mathbf{X}_{2\times3}\mathbf{X}^{\dagger}_{3\times2} = \mathbf{I}_{2\times2}$

Solving System of Linear Eq's

- Suppose we have a dataset of house prices where the price of a house is believed to be dependent on the size of the house (in square feet) and the number of bedrooms it has. We can model this relationship using a linear equation:

$$Price = w_0 + w_1 \times (Size) + w_2 \times (Bedrooms)$$

- Given the following data, determine $w_i, i = 0,1,2$

House Number	Size (sq.ft.)	Bedrooms	Price
House 1	1500	3	$300,000
House 2	2000	4	$400,000
House 3	1800	3	$330,000

$$300000 = w_0 + w_1 \times 1500 + w_2 \times 3$$
$$400000 = w_0 + w_1 \times 2000 + w_2 \times 4$$
$$330000 = w_0 + w_1 \times 1800 + w_2 \times 3$$

- $$\begin{bmatrix} 1 & 1500 & 3 \\ 1 & 2000 & 4 \\ 1 & 1800 & 3 \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix} = \begin{bmatrix} 300000 \\ 400000 \\ 330000 \end{bmatrix} \Rightarrow \begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix} = \begin{bmatrix} 1 & 1500 & 3 \\ 1 & 2000 & 4 \\ 1 & 1800 & 3 \end{bmatrix}^{-1} \begin{bmatrix} 300000 \\ 400000 \\ 330000 \end{bmatrix}$$

Solving System of Linear Eq's

```
A = np.array([[1, 1500, 3],
              [1, 2000, 4],
              [1, 1800, 3]])
B = np.array([300000, 400000, 330000]).reshape(-1, 1)
x = np.linalg.inv(A)@B
print(x)
```

```
[[    0.]
 [  100.]
 [50000.]]
```

Price=100×(Size)+50000×(Bedrooms)

Least Square Linear Regression (1)

- Repeat for the following data (7 houses):

House Number	Size (sq.ft.)	Bedrooms	Price
House 1	1500	3	$300,000
House 2	2000	4	$400,000
House 3	1800	3	$330,000
House 4	2200	4	$440,000
House 5	1600	3	$310,000
House 6	2400	5	$480,000
House 7	1700	3	$320,000

$$\begin{bmatrix} 1 & 1500 & 3 \\ 1 & 2000 & 4 \\ 1 & 1800 & 3 \\ 1 & 2200 & 4 \\ 1 & 1600 & 3 \\ 1 & 2400 & 5 \\ 1 & 1700 & 3 \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix} = \begin{bmatrix} 300,000 \\ 400,000 \\ 330,000 \\ 440,000 \\ 310,000 \\ 480,000 \\ 320,000 \end{bmatrix}$$

$$Ax = B$$

$$x = (A^T A)^{-1} A^T$$

Least Square Linear Regression (2)

```
A = np.array([
    [1, 1500, 3],
    [1, 2000, 4],
    [1, 1800, 3],
    [1, 2200, 4],
    [1, 1600, 3],
    [1, 2400, 5],
    [1, 1700, 3]])

B=np.array([
    [300_000],
    [400_000],
    [330_000],
    [440_000],
    [310_000],
    [480_000],
    [320_000]])

D = np.linalg.inv(A.T@A)@A.T@B
D = np.linalg.pinv(A)@B
estimated_prices = A @ D

print(D,np.hstack([B,estimated_prices]),sep="\n")
```

```
[[-14400.]
 [   144.]
 [ 31200.]]
[[300000. 295200.]
 [400000. 398400.]
 [330000. 338400.]
 [440000. 427200.]
 [310000. 309600.]
 [480000. 487200.]
 [320000. 324000.]]
```

Price=−14,400+144×(Size)+21,200×(Bedrooms)

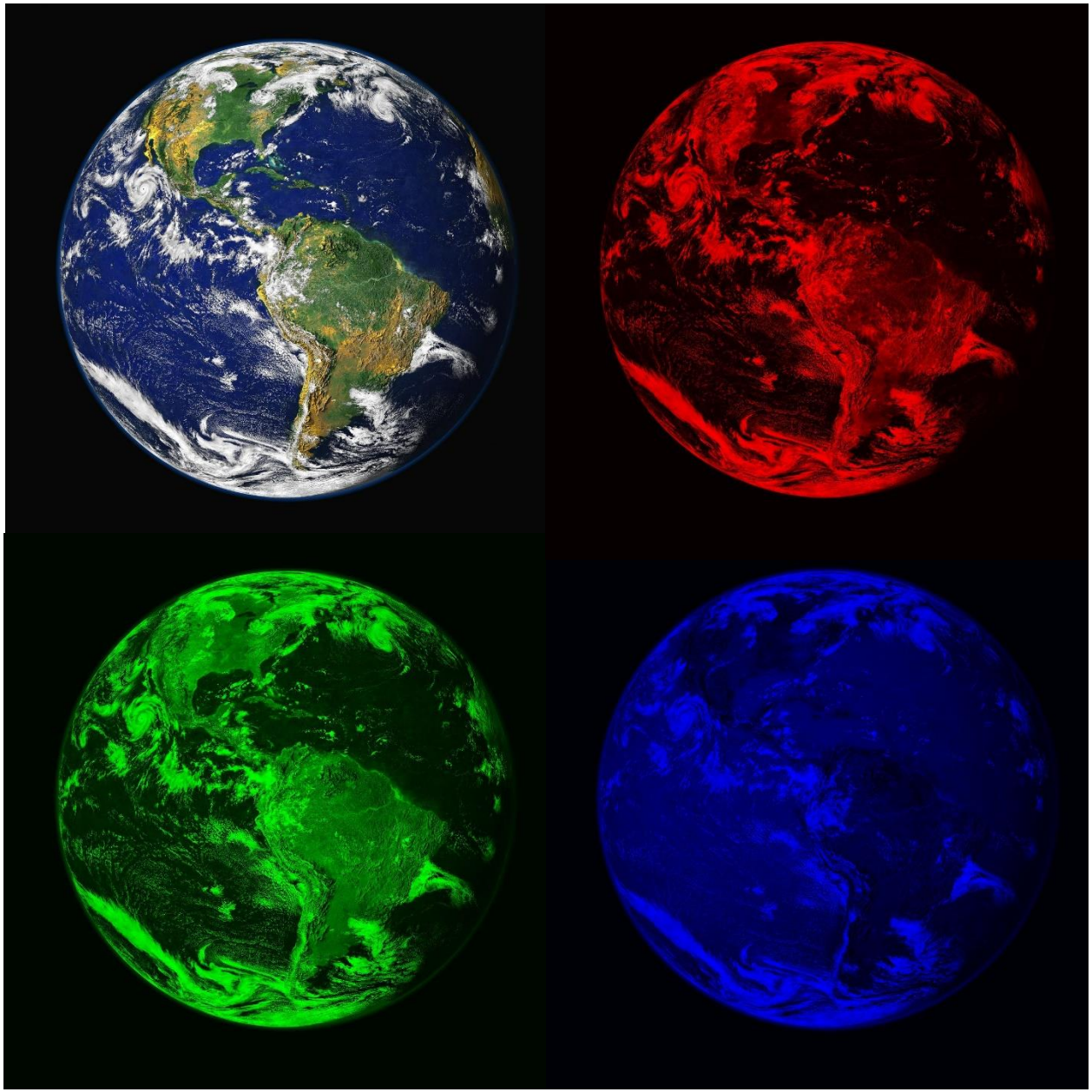
Tensors

- ***Tensors*** are n-dimensional arrays of scalars
  - Vectors are first-order tensors, $\mathbf{v} \in \mathbb{R}^n$
  - Matrices are second-order tensors, $\mathbf{A} \in \mathbb{R}^{m \times n}$
  - E.g., a fourth-order tensor is $\mathbf{T} \in \mathbb{R}^{n_1 \times n_2 \times n_3 \times n_4}$
- Tensors are denoted with upper-case letters of a special font face (e.g., $\mathbf{X}$, $\mathbf{Y}$, $\mathbf{Z}$)
- RGB images are third-order tensors, i.e., as they are 3-dimensional arrays
  - The 3 axes correspond to width, height, and channel
  - The channel axis corresponds to the color channels (red, green, and blue)

Image Processing (1)

```
from PIL import Image
import numpy as np

# Open an image file
with Image.open('Earth.jpg') as img:
    # Convert image to NumPy array
    img_array = np.array(img)


# Extract RGB channels
red_channel = img_array.copy()
red_channel[:, :, 1] = 0
red_channel[:, :, 2] = 0

green_channel = img_array.copy()
green_channel[:, :, 0] = 0
green_channel[:, :, 2] = 0

blue_channel = img_array.copy()
blue_channel[:, :, 0] = 0
blue_channel[:, :, 1] = 0

# Save each channel as a separate image
Image.fromarray(red_channel).save('earth_red_channel.jpg')
Image.fromarray(green_channel).save('earth_green_channel.jpg')
Image.fromarray(blue_channel).save('earth_blue_channel.jpg')
```

Image Processing (2)



Eigendecomposition

- ***Eigendecomposition*** is decomposing a matrix into a set of eigenvalues and eigenvectors
- ***Eigenvalues*** of a square matrix $\mathbf{A}$ are scalars λ and ***eigenvectors*** are any non-zero vectors $\mathbf{v}$ that satisfy

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$$

- Eigenvalues are found by solving the following equation

$$\det(\mathbf{A} - \lambda\mathbf{I}) = 0$$

- If a matrix $\mathbf{A}$ has n linearly independent eigenvectors $\{\mathbf{v}^1, \ldots, \mathbf{v}^n\}$ with corresponding eigenvalues $\{\lambda_1, \ldots, \lambda_n\}$, the eigendecomposition of $\mathbf{A}$ is given by

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

  - Columns of the matrix $\mathbf{V}$ are the eigenvectors, i.e., $\mathbf{V} = [\mathbf{v}^1, \ldots, \mathbf{v}^n]$
  - $\boldsymbol{\Lambda}$ is a diagonal matrix of the eigenvalues, i.e., $\boldsymbol{\Lambda} = [\lambda_1, \ldots, \lambda_n]$
- To find the inverse of the matrix A, we can use $\mathbf{A}^{-1} = \mathbf{V}\boldsymbol{\Lambda}^{-1}\mathbf{V}^{-1}$
  - This involves simply finding the inverse $\boldsymbol{\Lambda}^{-1}$ of a diagonal matrix

Eigendecomposition: Example

- ***Example:*** $\begin{bmatrix} 4 & 1 \\ 2 & 3 \end{bmatrix}$

Eigendecomposition: Example

```python
A = np.array([[4, 1], [2, 3]])

# Compute eigenvalues and eigenvectors
eigenvalues, eigenvectors = np.linalg.eig(A)

# Print the results
print("Eigenvalues:", eigenvalues)
print("Eigenvectors:", eigenvectors)

Lambda = np.diag(eigenvalues)

print(eigenvectors@Lambda@np.linalg.inv(eigenvectors))

print(eigenvectors@np.diag(1/eigenvalues)@np.linalg.inv(eigenvectors))
```

```
Eigenvalues: [5. 2.]
Eigenvectors: [[ 0.70710678 -0.4472136 ]
 [ 0.70710678  0.89442719]]
[[4. 1.]
 [2. 3.]]
[[ 0.3 -0.1]
 [-0.2  0.4]]
```

Eigendecomposition

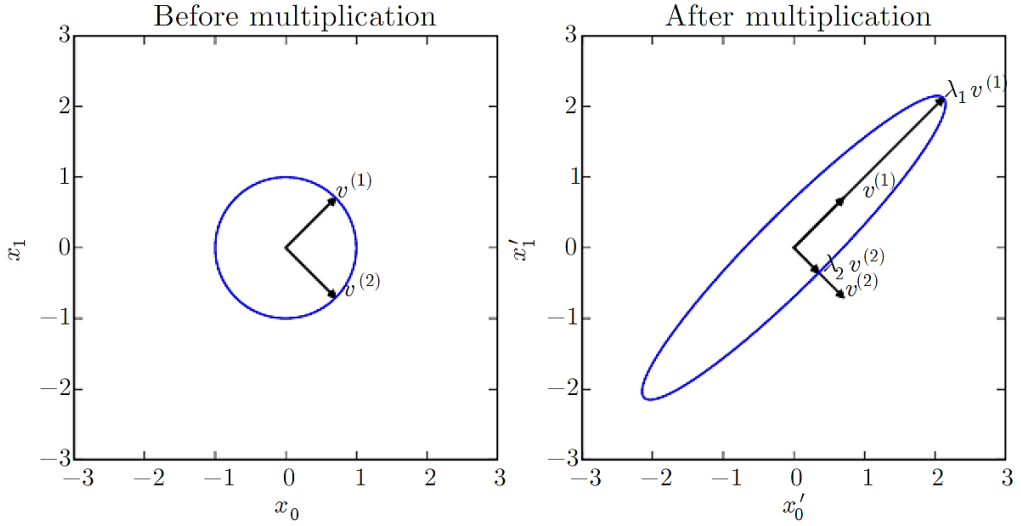
- Decomposing a matrix into eigenvalues and eigenvectors allows to analyze certain properties of the matrix
  - If all eigenvalues are positive, the matrix is positive definite
  - If all eigenvalues are positive or zero-valued, the matrix is positive semidefinite
  - If all eigenvalues are negative or zero-values, the matrix is negative semidefinite
    - Positive semidefinite matrices are interesting because they guarantee that $\forall \mathbf{x}, \mathbf{x}^T\mathbf{A}\mathbf{x} \geq 0$
- Eigendecomposition can also simplify many linear-algebraic computations
  - The determinant of A can be calculated as

$$\det(\mathbf{A}) = \lambda_1 \cdot \lambda_2 \cdots \lambda_n$$

  - If any of the eigenvalues are zero, the matrix is singular (it does not have an inverse)
- However, not every matrix can be decomposed into eigenvalues and eigenvectors
  - Also, in some cases the decomposition may involve complex numbers
  - Still, every real symmetric matrix is guaranteed to have an eigendecomposition according to $\mathbf{A} = \mathbf{V}\boldsymbol{\Lambda}\mathbf{V}^{-1}$, where $\mathbf{V}$ is an orthogonal matrix

Eigendecomposition

- Geometric interpretation of the eigenvalues and eigenvectors is that they allow to stretch the space in specific directions
  - Left figure: the two eigenvectors $\mathbf{v}^1$ and $\mathbf{v}^2$ are shown for a matrix, where the two vectors are unit vectors (i.e., they have a length of 1)
  - Right figure: the vectors $\mathbf{v}^1$ and $\mathbf{v}^2$ are multiplied with the eigenvalues λ_1 and λ_2
    - We can see how the space is scaled in the direction of the larger eigenvalue λ_1
- E.g., this is used for dimensionality reduction with PCA (principle component analysis) where the eigenvectors corresponding to the largest eigenvalues are used for extracting the most important data dimensions



Picture from: Goodfellow (2017) – Deep Learning

Principal Components Analysis (1)

- PCA is a dimensionality reduction technique. Transforms data into a new coordinate system.
- Applications: Data visualization, Noise reduction, Feature extraction in machine learning.
- Principal Components: Directions in which the data varies the most. Orthogonal to each other.
- Explained Variance: How much variance is captured by each principal component.
- Cumulative Explained Variance: Helps choose the number of components to retain.

Principal Components Analysis (2)

- Algorithm:

1. Compute the covariance matrix: $\Sigma = \frac{1}{n-1} M^T M$, where M is the mean-centered data matrix
2. Compute the correlation matrix: $\mathbf{R} = \big(\text{diag}(\Sigma)\big)^{-\frac{1}{2}} \Sigma \big(\text{diag}(\Sigma)\big)^{-\frac{1}{2}}$, where diag(Σ)is the matrix of diagonal elements of Σ
3. Take the eigendecomposition of **R**
4. 3. Sort the eigenvalues descending by magnitude, and sort the eigenvectors accordingly. Eigenvalues of the PCA are sometimes called latent factor scores.
5. Compute the "component scores" as the weighted combination of all data features, where the eigenvector provides the weights. The eigenvector associated with the largest eigenvalue is the "most important" component, meaning the one with the largest variance.
6. Convert the eigenvalues to percent variance explained to facilitate interpretation.

Principal Components Analysis (3)

- Suppose a bank collects the following eight variables for loan applicants: Income, Education, Age, Residency at current address, Years at current employer, Savings, Debt, and the number of credit cards. Using PCA, we want to reduce the number of dimensions.

Income	Education	Age	Residence	Employ	Savings	Debt	Credit cards
50000	16	28	2	2	5000	1200	2
72000	18	35	10	8	12000	5400	4
61000	18	36	6	5	15000	1000	2
88000	20	35	4	4	980	1100	4
91100	18	38	8	9	20000	0	1
45100	14	41	15	14	3900	22000	4
36200	14	29	6	5	100	7000	5
41000	12	34	9	8	5000	200	3
40000	16	32	8	7	19000	1760	2
32000	16	30	2	2	16000	550	1
29000	16	28	1	4	2100	4600	2
21240	12	26	2	2	100	10010	3
58700	12	38	9	9	4500	7800	5
41000	14	29	5	4	300	10000	6
38720	16	36	11	11	24500	540	2
88240	16	38	13	12	13600	8100	2
40000	18	39	7	6	16000	1300	2
34600	16	40	14	12	34000	100	3
29800	12	27	1	3	100	10000	5
56400	16	30	2	1	3000	1200	2
39800	14	29	3	2	2500	900	3
54200	16	31	5	3	14200	800	2
42650	16	27	3	2	5200	1000	3
62200	14	40	8	10	10000	700	2
72200	16	34	5	4	12000	400	4
26530	12	30	1	2	0	12000	2
36500	16	26	2	2	3100	800	3
40000	16	29	3	2	1900	1300	3
41200	12	34	5	4	1000	1200	2
50000	16	35	8	6	4500	1400	2

```
data = np.array([
    [50000, 16, 28, 2, 2, 5000, 1200, 2],
    [72000, 18, 35, 10, 8, 12000, 5400, 4],
    [61000, 18, 36, 6, 5, 15000, 1000, 2],
    [88000, 20, 35, 4, 4, 980, 1100, 4],
    [91100, 18, 38, 8, 9, 20000, 0, 1],
    [45100, 14, 41, 15, 14, 3900, 22000, 4],
    [36200, 14, 29, 6, 5, 100, 7000, 5],
    [41000, 12, 34, 9, 8, 5000, 200, 3],
    [40000, 16, 32, 8, 7, 19000, 1760, 2],
    [32000, 16, 30, 2, 2, 16000, 550, 1],
    [29000, 16, 28, 1, 4, 2100, 4600, 2],
    [21240, 12, 26, 2, 2, 100, 10010, 3],
    [58700, 12, 38, 9, 9, 4500, 7800, 5],
    [41000, 14, 29, 5, 4, 300, 10000, 6],
    [38720, 16, 36, 11, 11, 24500, 540, 2],
    [88240, 16, 38, 13, 12, 13600, 8100, 2],
    [40000, 18, 39, 7, 6, 16000, 1300, 2],
    [34600, 16, 40, 14, 12, 34000, 100, 3],
    [29800, 12, 27, 1, 3, 100, 10000, 5],
    [56400, 16, 30, 2, 1, 3000, 1200, 2],
    [39800, 14, 29, 3, 2, 2500, 900, 3],
    [54200, 16, 31, 5, 3, 14200, 800, 2],
    [42650, 16, 27, 3, 2, 5200, 1000, 3],
    [62200, 14, 40, 8, 10, 10000, 700, 2],
    [72200, 16, 34, 5, 4, 12000, 400, 4],
    [26530, 12, 30, 1, 2, 0, 12000, 2],
    [36500, 16, 26, 2, 2, 3100, 800, 3],
    [40000, 16, 29, 3, 2, 1900, 1300, 3],
    [41200, 12, 34, 5, 4, 1000, 1200, 2],
    [50000, 16, 35, 8, 6, 4500, 1400, 2]])
```

https://statisticsbyjim.com/basics/principal-component-analysis/

Principal Components Analysis (4)

```
# Mean-center data
data_cent = data - np.mean(data,axis = 0)
# Calculate the covariance matrix
cov_mat = 1/(data.shape[0]-1)*data_cent.T@data_cent
D = np.diag(np.diag(cov_mat))
# Calculate the correlation matrix
cor_mat =np.sqrt(np.linalg.inv(D))@cov_mat@np.sqrt(np.linalg.inv(D))
# np.corrcoef(data_cent, rowvar = False)
np.set_printoptions(precision=2, suppress=True)
# Eigendecomposition
L, V = np.linalg.eigh(cor_mat)
# Sort eigenvalues (and corresponding vectors) descending
sorted_indices = np.argsort(L)[::-1]
L = L[sorted_indices]
V = V[:, sorted_indices]
# Ratio of each eignevalue to total
r = L/np.sum(L)
projected_data = np.dot(data_cent, V)
```

Principal Components Analysis (5)

Income
Education
Age
Residence
Employ
Savings
Debt
Credit cards

✓ L ···

```
array([3.55, 2.13, 1.04, 0.53, 0.41, 0.17, 0.13, 0.04])
```

✓ V ···

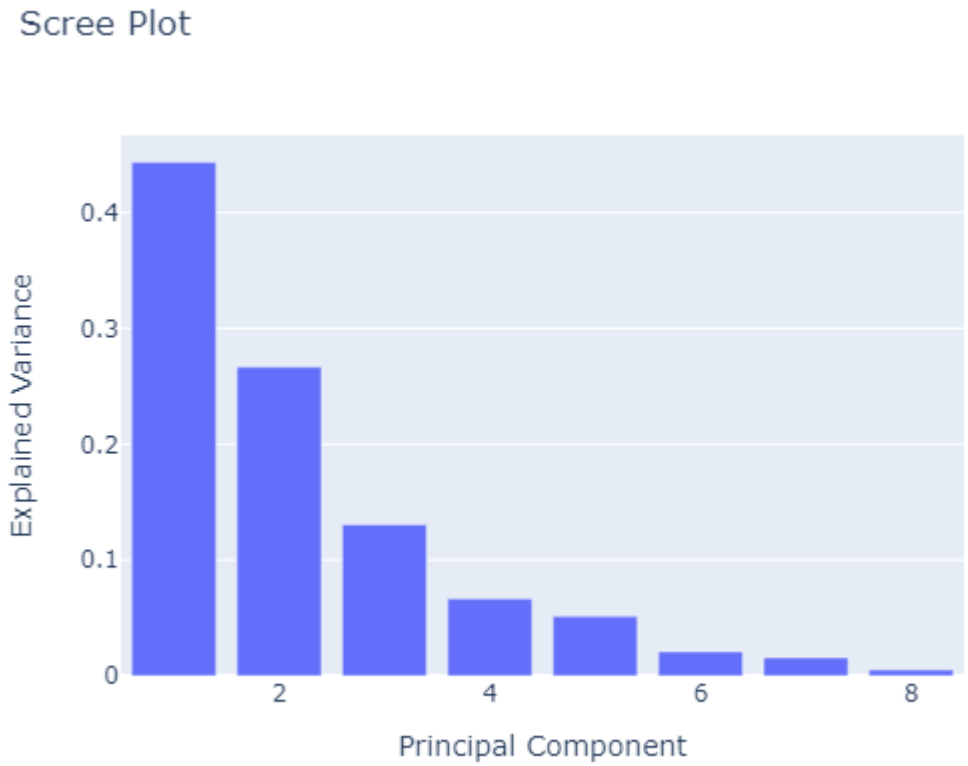
```
array([[-0.31,  0.14,  0.68, -0.35,  0.24,  0.49, -0.02,  0.03],
       [-0.24,  0.44,  0.4 ,  0.24, -0.62, -0.36, -0.1 , -0.06],
       [-0.48, -0.14,  0.  , -0.21,  0.17, -0.49,  0.66,  0.05],
       [-0.47, -0.28, -0.09,  0.12,  0.04, -0.09, -0.49,  0.66],
       [-0.46, -0.3 , -0.12, -0.02,  0.01, -0.02, -0.37, -0.74],
       [-0.4 ,  0.22, -0.37,  0.44, -0.14,  0.57,  0.35,  0.02],
       [ 0.07, -0.59,  0.08, -0.28, -0.68,  0.25,  0.2 ,  0.07],
       [ 0.12, -0.45,  0.47,  0.7 ,  0.19, -0.02,  0.16, -0.06]])
```

✓ r ···

```
array([0.44, 0.27, 0.13, 0.07, 0.05, 0.02, 0.02, 0.01])
```

Principal Components Analysis (6)

```
import plotly.express as px
import numpy as np
fig = px.bar(x=range(1,9),
             y=r,
             labels={'x':'Principal Component', 'y':'Explained Variance'},
             title='Scree Plot')
fig.show()
```



Network Analysis (1)

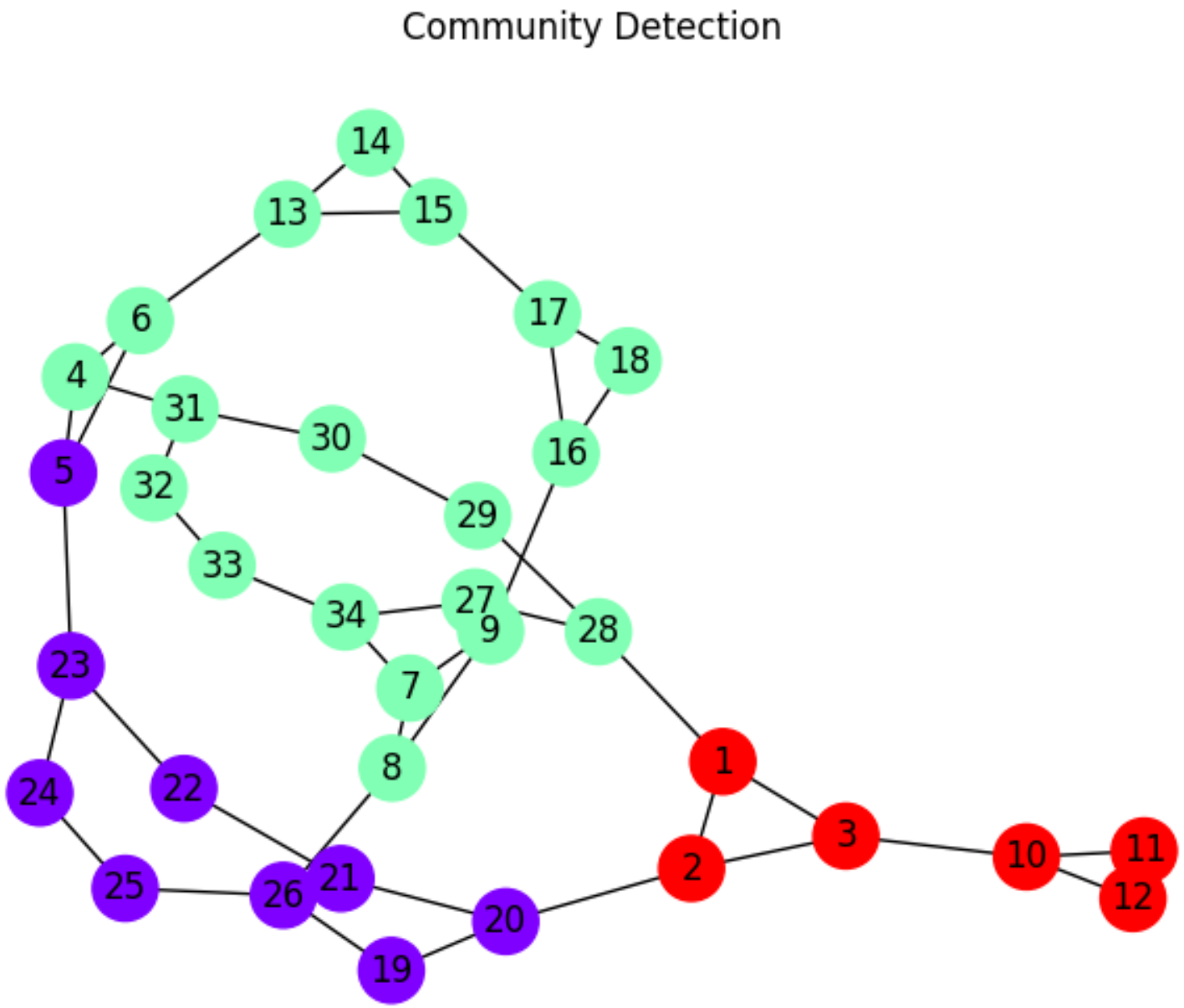
- Eigendecomposition, specifically through techniques like spectral clustering and the use of Laplacian matrices, is often used in the analysis of social networks to identify communities or clusters within the network, analyze the importance of nodes (individuals or entities in the network), and understand the overall structure and properties of the network.
- Example Steps for Community Detection (Spectral Clustering):

1. Construct the Adjacency Matrix $\mathbf{A}$: For a social network, each node represents an individual or entity, and each edge represents a relationship between two nodes ($A_{ij} = 1$, if these two nodes are connected).
2. Compute the Degree Matrix $\mathbf{D}$: A diagonal matrix where each diagonal element is the sum of the i-th row of $\mathbf{A}$
3. Calculate the Laplacian Matrix $\mathbf{L} = \mathbf{A} - \mathbf{D}$: Often, the normalized Laplacian is used.
4. Eigendecomposition: Find the eigenvalues and eigenvectors of the Laplacian.
5. Select Eigenvectors: Use the eigenvectors corresponding to the smallest non-zero eigenvalues to form a matrix.
6. Cluster Rows: Treat each row of the matrix formed by selected eigenvectors as a point in a low-dimensional space, and cluster them using a technique like k-means. Each cluster represents a community in the network

Network Analysis (2)

```python
import networkx as nx
import numpy as np
import sklearn
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt

# Step 1: Construct the Adjacency Matrix
# Create a sample graph (you should replace this with your actual graph)
G = nx.Graph()
edges = [
    (1, 2), (2, 3), (3, 1), (4, 5), (5, 6), (6, 4), (7, 8), (8, 9), (9, 7),(10, 11), (11, 12), (12, 10), (13, 14), (14, 15), (15, 13),
    (16, 17), (17, 18), (18, 16), (3, 10), (6, 13), (9, 16), (19, 20), (20, 21), (21, 22), (22, 23), (23, 24), (24, 25),
    (25, 26), (26, 19), (2, 20), (5, 23), (8, 26), (27, 28), (28, 29), (29, 30), (30, 31), (31, 32), (32, 33), (33, 34), (34, 27),
    (1, 28), (4, 31), (7, 34), (15,17) ]
G.add_edges_from(edges)
# Get the Adjacency Matrix (A)
A = nx.adjacency_matrix(G).toarray()
# Step 2: Calculate the Laplacian Matrix
# Degree matrix (D)
D = np.diag(A.sum(axis=1))
# Graph Laplacian (L)
L = D - A
# Normalized Laplacian (L_sym)
D_half_inv = np.linalg.inv(np.sqrt(D))
L_sym = np.dot(D_half_inv, np.dot(L, D_half_inv))
# Step 3: Eigendecomposition of the Laplacian
eigenvalues, eigenvectors = np.linalg.eigh(L_sym)
# Step 4: Select k eigenvectors corresponding to the k smallest non-zero eigenvalues
k = 3  # Set the number of communities you expect
selected_eigenvectors = eigenvectors[:, 1:k+1]
# Step 5: Cluster Rows using k-means
kmeans = KMeans(n_clusters=k)
kmeans.fit(selected_eigenvectors)
colors = kmeans.labels_
# Visualization (Optional)
pos = nx.spring_layout(G)
nx.draw(G, pos, with_labels=True, node_color=colors, node_size=500, cmap=plt.cm.rainbow)
plt.title("Community Detection")
plt.show()
```

Network Analysis (3)



Singular Value Decomposition

- ***Singular value decomposition*** (SVD) provides another way to factorize a matrix, into singular vectors and singular values
  - SVD is more generally applicable than eigendecomposition
  - Every real matrix has an SVD, but the same is not true of the eigendecomposition
    - E.g., if a matrix is not square, the eigendecomposition is not defined, and we must use SVD
- SVD of an $m \times n$ matrix $\mathbf{A}$ is given by

$$\mathbf{A} = \mathbf{U}\mathbf{D}\mathbf{V}^{\boldsymbol{T}}$$

  - $\mathbf{U}$ is an $m \times m$ matrix, $\mathbf{D}$ is an $m \times n$ matrix, and $\mathbf{V}$ is an $n \times n$ matrix
  - The elements along the diagonal of $\mathbf{D}$ are known as the singular values of A
  - The columns of $\mathbf{U}$ are known as the left-singular vectors
  - The columns of $\mathbf{V}$ are known as the right-singular vectors
- One of the most useful features of the SVD is that we can use it to derive a pseudo-inverse of non-square matrices
- The rank of the original matrix $\mathbf{A}$ is equal to the number of non-zero singular values
- For a square matrix, the singular values of $\mathbf{A}$ are the square roots of the eigenvalues of $\mathbf{A}^{\mathbf{T}}\mathbf{A}$ or $\mathbf{A}\mathbf{A}^{\mathbf{T}}$

Singular Value Decomposition

```
A = np.array([[1, 2],
              [3, 4],
              [5, 6]])
print("Original Matrix A:")
print(A)
U, Sigma, VT = np.linalg.svd(A, full_matrices=True)
print("Matrix U (Left Singular Vectors):")
print(U)
print("\nDiagonal Matrix Sigma (Singular Values):")
print(Sigma)
print("\nMatrix VT (Right Singular Vectors):")
print(VT)
# Convert Sigma to the appropriate dimensional matrix
Sigma_mat = np.zeros((A.shape[0], A.shape[1]))
for i in range(min(A.shape)):
    Sigma_mat[i, i] = Sigma[i]
# Reconstruct the original matrix
A_reconstructed = U @ Sigma_mat @ VT
print("Reconstructed Matrix A:")
print(A_reconstructed)
# Relationship with eigenvalues
L, V = np.linalg.eig(A.T@A)
print(f"Square root of eigval of A.T@A: {np.sqrt(L)}")
```

```
Original Matrix A:
[[1 2]
 [3 4]
 [5 6]]
Matrix U (Left Singular Vectors):
[[-0.23  0.88  0.41]
 [-0.52  0.24 -0.82]
 [-0.82 -0.4   0.41]]

Diagonal Matrix Sigma (Singular Values):
[9.53 0.51]

Matrix VT (Right Singular Vectors):
[[-0.62 -0.78]
 [-0.78  0.62]]
Reconstructed Matrix A:
[[1. 2.]
 [3. 4.]
 [5. 6.]]
Square root of eigval of A.T@A: [0.51 9.53]
```

Image Compression using SVD (1)

- Apply SVD to image matrix (or to each channel separately if RGB). Then choose the *k* largest singular values and associated singular vectors and reconstruct compressed image matrix.

```
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt

# Step 1: Read and Prepare the Image
image = Image.open(r'chipmunk.jpg')  # Replace with your image path

image_array = np.array(image) / 255.0  # Convert to numpy array and normalize to [0, 1]

# Step 2: Apply SVD on Each Channel
def compress_channel(channel, k):
    U, Sigma, VT = np.linalg.svd(channel, full_matrices=False)
    U_k = U[:, :k]
    Sigma_k = np.diag(Sigma[:k])
    VT_k = VT[:k, :]
    return U_k @ Sigma_k @ VT_k
k = 20  # Number of principal components to keep
reconstructed_channels = [compress_channel(image_array[:, :, i], k) for i in range(image_array.shape[2])]

# Step 3: Reconstruct the Image
reconstructed_image = np.stack(reconstructed_channels, axis=2)

# Clip values to be in valid range [0, 1]
reconstructed_image = np.clip(reconstructed_image, 0, 1)

# Step 4: Save or Display the Compressed Image
plt.imshow(reconstructed_image)
plt.title(f'Image with {k} Principal Components')
plt.axis('off')
plt.show()

reconstructed_image_pil = Image.fromarray((reconstructed_image*255).astype(np.uint8))
output_path = r'chipmunk3.jpg'
reconstructed_image_pil.save(output_path, 'JPEG')
```

Image Compression using SVD (2)

Original

K=20

K=10