



# CS316 INTRODUCTION TO AI AND DATA SCIENCE

## CHAPTER 4 Integrating Linear Algebra with NumPy: Practical Foundations for Data Science

### LECTURE 1 *Vectors, Matrices, Dot Product Normalization and Cosine Similarity*

Prof. Anis Koubaa

SEP 2024

# What is NumPy?

- **NumPy Overview**

- A fundamental Python library for numerical computing.
- Provides support for large, multi-dimensional arrays and matrices.
- Includes a collection of mathematical functions to operate on these arrays.

- **Why NumPy?**

- **Performance:** Implemented in C, providing fast execution.
- **Functionality:** Includes functions for linear algebra, Fourier transform, and random number generation.
- **Integration:** Forms the basis of most Python-based scientific computing solutions.

python

```
import numpy as np
# Creating a simple NumPy array
a = np.array([1, 2, 3, 4, 5])
print(a)
```

Array [1 2 3 4 5]

dtype=int64

# Why Use NumPy?

- **Advantages of NumPy**

- Optimized for numerical operations with N-dimensional array support.
- Essential for scientific computing with Python.
- Facilitates complex mathematical functions and operations.

- **Powerful Computing Capabilities**

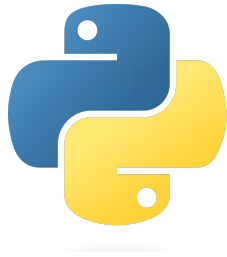
- Efficient operations with arrays and matrices.
- Supports broadcasting and advanced indexing techniques.
- Integrates C/C++ and Fortran code for high performance.

python

```
import numpy as np
# Creating a simple NumPy array
a = np.array([1, 2, 3, 4, 5])
print(a)
```

Array [1 2 3 4 5]

dtype=int64



# Install Numpy



**CS316**  
**INTRODUCTION TO AI AND DATA  
SCIENCE**

**CHAPTER 3**  
**PANDAS AND NUMPY**

**LECTURE 1**

*PANDAS*

# How to Install NumPy?

- **Installation Guide**

- **Using pip (Python's Package Installer):**

- Ensure Python and pip are already installed on your system.
    - Open your command prompt (Windows) or terminal (Mac/Linux).
    - Type the following command and press Enter:

```
bash
```

[Copy code](#)

```
pip install numpy
```

- **Using Anaconda:**

- Anaconda is a popular Python distribution for data science and machine learning that includes NumPy.
    - If Anaconda is installed, NumPy can be installed via the Anaconda Prompt:

```
bash
```

[Copy code](#)


```
conda install numpy
```

# How to Install NumPy?

## Verifying Installation

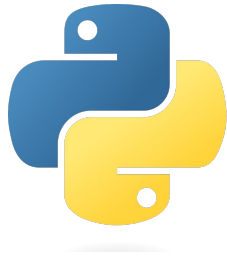
- To ensure NumPy is installed correctly, run:

python

 Copy code

```
import numpy as np
print(np.__version__)
```

This command will print the installed version of NumPy.



# Scalars and NumPy Basics



**CS316**  
**INTRODUCTION TO AI AND DATA  
SCIENCE**

**CHAPTER 3**  
**PANDAS AND NUMPY**

**LECTURE 1**

*PANDAS*

# Understanding Scalars with Numpy

- **Linear Algebra Concept:**
  - **Definition of Scalars:** Scalars are single numerical values that can represent quantities like length or temperature.
  - **Operations with Scalars:** Common operations include addition, subtraction, multiplication, and division.
- **NumPy Implementation:**
  - **Creating Scalar Variables:**

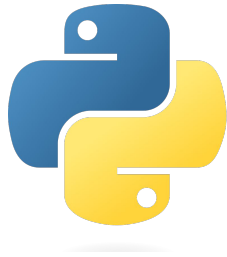
```
python Copy code  
  
import numpy as np  
x = np.array(5) # x is a scalar in NumPy
```

- **Performing Arithmetic Operations with NumPy Scalars:**

```
python Copy code  
  
y = x - 2    # Subtraction  
z = x * 3    # Multiplication  
w = x / 2    # Division  
print("Subtract 2:", y)  
print("Multiply by 3:", z)  
print("Divide by 2:", w)
```

## Using Scalars in NumPy:

- Scalars in NumPy behave similarly to numbers in basic Python but are optimized for performance when used in array operations.
- They automatically integrate with NumPy's universal functions for array-based computing.



# Numpy Arrays Basics



**CS316**  
**INTRODUCTION TO AI AND DATA  
SCIENCE**

**CHAPTER 3**  
**PANDAS AND NUMPY**

**LECTURE 1**

*PANDAS*

# Understanding Scalars with Numpy

- 2.2 NumPy Array Basics

- Key Concepts:

- Python Lists vs. NumPy Arrays:

- Lists: Flexible, slow for large datasets.
      - Arrays: Optimized for speed, designed for numerical operations.

- Advantages of NumPy Arrays:

- Faster data handling.
    - Built-in mathematical functions.
    - Efficient memory usage.

- Practical Application:

- Creating and Examining Arrays:

```
python Copy code  
  
import numpy as np  
arr1 = np.array([1, 2, 3, 4, 5]) # One-dimensional array  
arr2 = np.array([[1, 2, 3], [4, 5, 6]]) # Two-dimensional array  
# Attributes: shape, dtype, ndim  
print("Shape:", arr1.shape, "| Dtype:", arr1.dtype, "| Ndim:", arr2.ndim)
```

# Shape | Data Type | Dimension

python

Copy code

```
import numpy as np

# Creating a one-dimensional array
arr1 = np.array([1, 2, 3, 4, 5])
# Creating a two-dimensional array
arr2 = np.array([[1, 2, 3], [4, 5, 6]])

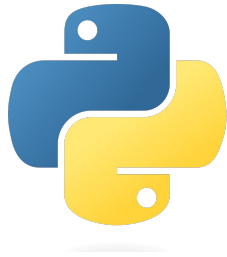
# Displaying attributes of arr1
print("Attributes of arr1:")
print("Shape:", arr1.shape) # Outputs the shape of the array
print("Data Type:", arr1.dtype) # Outputs the data type of the array elements
print("Number of Dimensions:", arr1.ndim) # Outputs the number of dimensions

# Displaying attributes of arr2
print("\nAttributes of arr2:")
print("Shape:", arr2.shape) # Outputs the shape of the array
print("Data Type:", arr2.dtype) # Outputs the data type of the array elements
print("Number of Dimensions:", arr2.ndim) # Outputs the number of dimensions
```

The execution results for the NumPy array code are as follows:

- Attributes of `arr1` (One-dimensional array):
  - Shape: (5,)
  - Data Type: int64
  - Number of Dimensions: 1
- Attributes of `arr2` (Two-dimensional array):
  - Shape: (2, 3)
  - Data Type: int64
  - Number of Dimensions: 2

```
(['Attributes of arr1:',  
  'Shape: (5,)',  
  'Data Type: int64',  
  'Number of Dimensions: 1'],  
 ['\nAttributes of arr2:',  
  'Shape: (2, 3)',  
  'Data Type: int64',  
  'Number of Dimensions: 2'])
```



# Vectors & Numpy Arrays



**CS316**  
**INTRODUCTION TO AI AND DATA  
SCIENCE**

**CHAPTER 3**  
**PANDAS AND NUMPY**

**LECTURE 1**  
*PANDAS*

# 1D Vectors (Arrays)

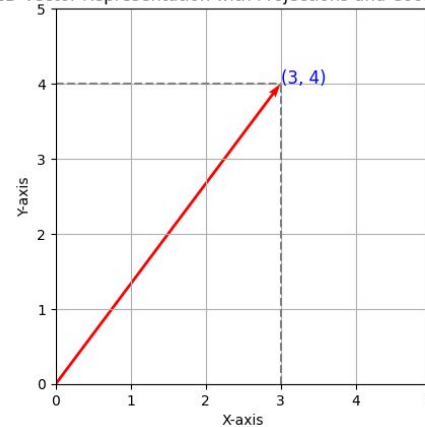
- **Linear Algebra Concept:**
  - **Definition of Vectors:** Vectors are denoted as  $\mathbf{v} = [v_1, v_2, \dots, v_n]$  in mathematics, representing an ordered collection of  $n$  elements, each element a coordinate in  $n$ -dimensional space.
- **Data Science Application:**
  - **Feature Vectors:** In data science, vectors are used as feature vectors, where each element  $v_i$  represents a distinct attribute or feature of a data point, crucial for models in AI and machine learning.
- **NumPy Implementation:**
  - **Representing Vectors Using 1D Arrays:**

```
python Copy code  
  
import numpy as np  
# Example of initializing a feature vector in NumPy  
feature_vector = np.array([5, 3, 1, 8])
```

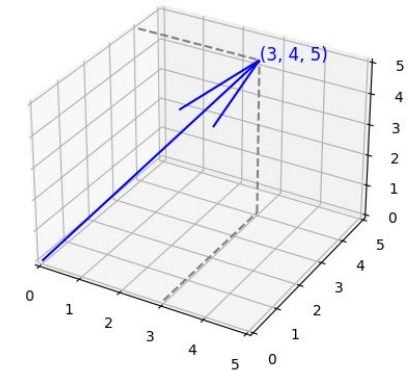
- **Initializing Vectors:**

```
python Copy code  
  
# Creating zero and one vectors commonly used in initialization  
zero_vector = np.zeros(4) # Vector of zeros  
one_vector = np.ones(4)   # Vector of ones
```

2D Vector Representation with Projections and Coordinates



3D Vector Representation with Projections and Coordinates



```
# Define a 2D and a 3D vector  
vector_2d = np.array([3, 4])  
vector_3d = np.array([3, 4, 5])
```

# Vector Operations

## 3.2 Vector Operations

- Operations in Linear Algebra:
  - Addition and Subtraction of Vectors:
    - $\mathbf{v} = [2, 4, 6]$
    - $\mathbf{w} = [1, 0, 1]$
    - Addition:  $\mathbf{v} + \mathbf{w} = [2 + 1, 4 + 0, 6 + 1] = [3, 4, 7]$
    - Subtraction:  $\mathbf{v} - \mathbf{w} = [2 - 1, 4 - 0, 6 - 1] = [1, 4, 5]$
  - Scalar Multiplication:
    - Scalar  $c = 3$
    - $c \cdot \mathbf{v} = 3 \cdot [2, 4, 6] = [6, 12, 18]$

NumPy Implementation:

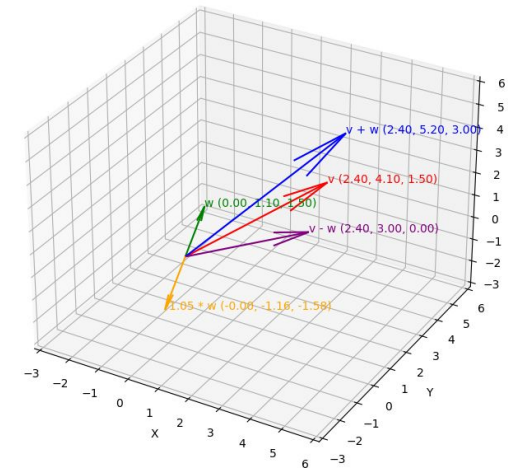
- Performing Vector Addition and Subtraction:

```
python Copy code  
  
import numpy as np  
v = np.array([2, 4, 6])  
w = np.array([1, 0, 1])  
addition = v + w # Vector addition  
subtraction = v - w # Vector subtraction
```

- Scalar Multiplication:

```
python Copy code  
  
scalar = 3  
scaled_vector = v * scalar # Scalar multiplication
```

Vector Operations: Addition, Subtraction, and Scalar Multiplication



# 3D Vector Operations Sum | Subtraction

## NumPy Implementation:

- Performing Vector Addition and Subtraction:

python

Copy code

```
import numpy as np
v = np.array([2, 4, 6])
w = np.array([1, 0, 1])
addition = v + w # Vector addition
subtraction = v - w # Vector subtraction
```

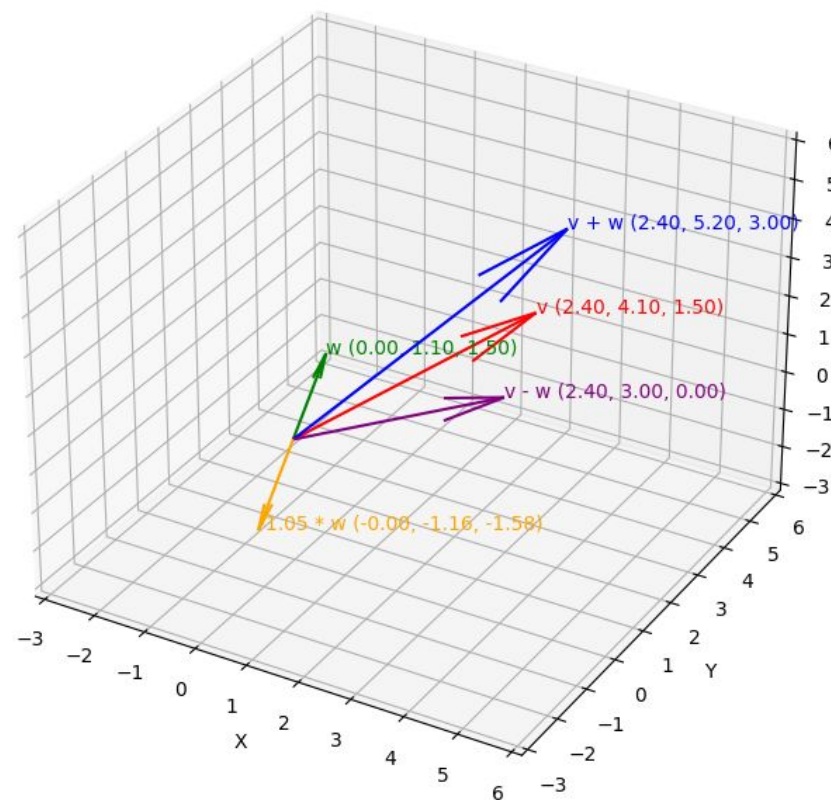
- Scalar Multiplication:

python

Copy code

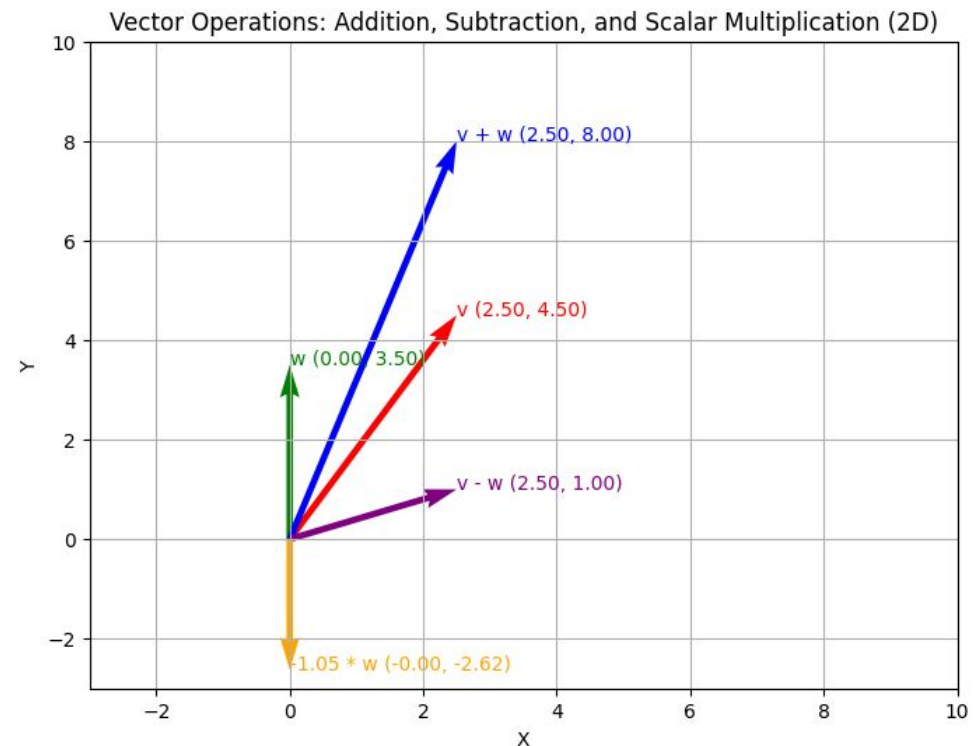
```
scalar = 3
scaled_vector = v * scalar # Scalar multiplication
```

Vector Operations: Addition, Subtraction, and Scalar Multiplication



# 2D Vector Operations – Sum | Subtraction

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 # Define the 2D vectors v and w
5 v = np.array([2.5, 4.5])
6 w = np.array([0.0, 3.5])
7
8 # Vector operations
9 addition = v + w
10 subtraction = v - w
11 c = -0.75
12 scalar_multiplication = c * w
13
```



# Application of Vectors in AI

## Vectors in Computer Vision:

- **Image Representation:**

- Images in computer vision are represented as vectors or matrices of pixel values.
- For example, a grayscale image can be represented as a 2D array where each element corresponds to the intensity of a pixel.
- **Math Example:** A  $2 \times 2$  grayscale image might be represented in NumPy as:

python

Copy code

```
import numpy as np
image = np.array([[0, 255], [255, 0]]) # 0 = Black, 255 = White
```

- This simple array shows how images are treated as numerical data, which AI models can process to recognize patterns or features.

Image shape: (275, 183, 3)

Image array:

```
[[[255 255 255]
  [255 255 255]
  [255 255 255]
  ...
  [255 255 255]
  [255 255 255]
  [255 255 255]]
```

```
[[[255 255 255]
  [255 255 255]
  [255 255 255]
  ...
  [255 255 255]
  [255 255 255]
  [255 255 255]]
```

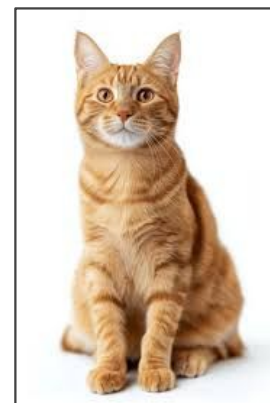
```
[[[255 255 255]
  [255 255 255]
  [255 255 255]
  ...
  [255 255 255]
  [255 255 255]
  [255 255 255]]
```

```
[[[255 255 255]
  [255 255 255]
  [255 255 255]
  ...
  [255 255 255]
  [255 255 255]
  [255 255 255]]
```

```
[[[255 255 255]
  [255 255 255]
  [255 255 255]
  ...
  [255 255 255]
  [255 255 255]
  [255 255 255]]
```

```
[[[255 255 255]
  [255 255 255]
  [255 255 255]
  ...
  [255 255 255]
  [255 255 255]
  [255 255 255]]
```

...



# Application of Vectors in NLP

## Vectors in Natural Language Processing (NLP):

- **Word Representation:**

- Words in NLP are represented as vectors in a high-dimensional space (word embeddings).
- These vectors capture semantic meanings where words with similar meanings are closer in the vector space.
- **Math Example:** Representing the word "king" as a vector might be abstractly visualized as:

python

Copy code

```
king = np.array([0.2, -0.4, 0.7, 0.3]) # Hypothetical vector coord
```

- This vector could be part of a model trained to perform operations like finding synonyms or analyzing text sentiment.

```
1 import numpy as np
2
3 # Let's assume we have 3 words and we're embedding them in a 5-dimensional vector
4 word_to_embedding = {
5     'apple': np.array([0.1, 0.3, 0.5, 0.7, 0.9]),
6     'banana': np.array([0.2, 0.4, 0.6, 0.8, 1.0]),
7     'cherry': np.array([0.3, 0.5, 0.7, 0.9, 1.1])
8 }
9
10 # Let's select a word to embed
11 word = 'apple'
12
13 # Get the word's embedding vector
14 embedding_vector = word_to_embedding[word]
15
16 # Display the embedding vector
17 print(f"Embedding vector for the word '{word}':")
18 print(embedding_vector)
19
20 # You can perform operations on these vectors, such as calculating similarity
21 # For example, calculating the dot product between "apple" and "banana"
22 similarity = np.dot(word_to_embedding['apple'], word_to_embedding['banana'])
23
24 print(f"\nSimilarity (dot product) between 'apple' and 'banana': {similarity}")
25
```

Embedding vector for the word 'apple':  
[0.1 0.3 0.5 0.7 0.9]

Similarity (dot product) between 'apple' and 'banana': 1.9

# Application of Vectors in NLP

```
1 import spacy
2 import numpy as np
3
4 # Load the small English model in spaCy
5 nlp = spacy.load("en_core_web_sm")
6
7 # Choose a word and convert it to a vector using spaCy
8 word = "apple"
9 doc = nlp(word)
10 embedding_vector = doc.vector
11
12 # Convert the spaCy vector to a NumPy array
13 embedding_vector = np.array(embedding_vector)
14
15 # Print the embedding vector
16 print(f"Embedding vector for the word '{word}':")
17 print(embedding_vector)
18
19 # Print the Shape of the vector
20 print(f"Shape of the embedding vector for the word '{word}' is {embedding_vector.shape}:")
21
```

Embedding vector for the word 'apple':

```
[ -1.2599487  -0.87038326 -1.0834986   0.5798379   0.03857595 -0.02588724
   0.9775548   0.31093895  0.19412561 -0.8062178   0.43808204 -1.8498268
  -0.30574286  0.5693637   0.42844394  0.69174224 -0.7197368  -1.2614795
   0.83457553  0.14667332  0.12171662  0.48029226  0.50147873 -0.4299112
   0.5533447   0.8749714   0.71914    0.5731143  -0.5064311   0.38493997
  -0.31778833  0.18084693  0.5162936  -0.00233826  0.1870515  -1.3773322
   1.009095   -0.10771251  1.6994228   0.78603184 -0.8166558   0.57896584
  -0.5232718   0.7045958  -0.46308953 -0.37629813 -0.38788998  0.1730735
  -0.05550597 -0.17245518  0.62919456  0.87473    0.60047954 -0.27686393
   0.8524152  -0.28676936  0.9972549  -0.71060055  0.11830124 -0.37214422
  -1.3039289  -0.02281845  0.4063236  -0.43118405  0.9401908  -0.02761412
  -0.39026427 -0.29733896  0.78710043 -0.34422576  0.11906591  0.8003473
   1.4978364  -0.38792044 -0.5264353  -0.38889915 -0.28553864 -0.22295064
   0.8420893  -0.79365766 -0.0956156  -0.96640915 -1.1665895  -1.1019065
  -0.602306   0.765056   0.3859367  -0.31349194 -1.237845   0.11333084
  -1.6053262   0.14791119  0.6127024   1.0456864   0.8747115   0.6120273 ]
```

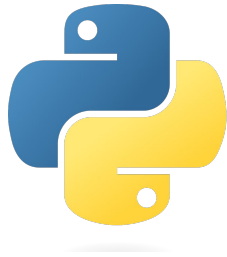
Shape of the embedding vector for the word 'apple' is (96,):

## Explanation:

1. We use the `en_core_web_sm` model from spaCy, which is a small pre-trained model for English.
2. For a word like `apple`, we get the word's vector (embedding) using spaCy's `doc.vector`.
3. We print the embedding vector and demonstrate how to compute the similarity (in this case, using the dot product) between two words (`apple` and `banana`).

## Notes:

- The `en_core_web_sm` model uses relatively simple embeddings, but larger models (like `en_core_web_md` or `en_core_web_lg`) provide richer embeddings with more detailed word representations.
- If you want to use more advanced embeddings, you can download larger models or use pre-trained models like GloVe or Word2Vec.



# Vectors & Numpy Arrays Dot Product



**CS316**  
**INTRODUCTION TO AI AND DATA  
SCIENCE**

**CHAPTER 3**  
**PANDAS AND NUMPY**

**LECTURE 1**  
*PANDAS*

# Dot Product

- **Linear Algebra Concept:**
  - **Dot Product:** The dot product of two vectors, feature vector  $\mathbf{x}$  and weight vector  $\mathbf{w}$ , is calculated as  $\mathbf{x} \cdot \mathbf{w} = \sum_{i=1}^n x_i w_i$ . This operation can be seen as a linear transformation of the features by the weights.
  - **Properties:**
    - **Commutative:**  $\mathbf{x} \cdot \mathbf{w} = \mathbf{w} \cdot \mathbf{x}$
    - **Distributive over addition**
    - **Scalar multiplication compatibility**

NumPy Implementation:

- **Computing Dot Product:**

```
python Copy code  
  
import numpy as np  
x = np.array([2, 4, 6]) # Feature vector  
w = np.array([1, 0, 1]) # Weight vector  
dot_product = np.dot(x, w) # Dot product using np.dot  
dot_product_operator = x @ w # Dot product using @ operator
```

# Dot Product

- **Definition:**

- The dot product of two vectors  $\mathbf{x}$  and  $\mathbf{y}$  is a scalar that represents the sum of the products of their corresponding components.

- **Compact Math Form:**

- $\mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^n x_i y_i$

- **Expanded Math Form:**

- Given vectors  $\mathbf{x} = [x_1, x_2, x_3]$  and  $\mathbf{y} = [y_1, y_2, y_3]$ , the dot product can be expressed as:

$$\mathbf{x} \cdot \mathbf{y} = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = x_1 \cdot y_1 + x_2 \cdot y_2 + x_3 \cdot y_3$$

- **Numerical Example:**

- For vectors  $\mathbf{x} = [2, 3, 4]$  and  $\mathbf{y} = [1, 0, -1]$ :

$$\mathbf{x} \cdot \mathbf{y} = 2 \cdot 1 + 3 \cdot 0 + 4 \cdot (-1) = 2 + 0 - 4 = -2$$

- **Purpose:**

- The dot product quantifies how much one vector extends in the direction of another, thus providing a measure of alignment between the vectors. It is widely used to measure vector similarity in various fields of mathematics and physics.

python

Copy code

```
import numpy as np

# Define the vectors
x = np.array([2, 3, 4])
y = np.array([1, 0, -1])

# Compute the dot product using NumPy
dot_product = np.dot(x, y)

# Print the results
print("Vectors x and y:")
print("x =", x)
print("y =", y)
print("Dot Product (x · y) =", dot_product)
```

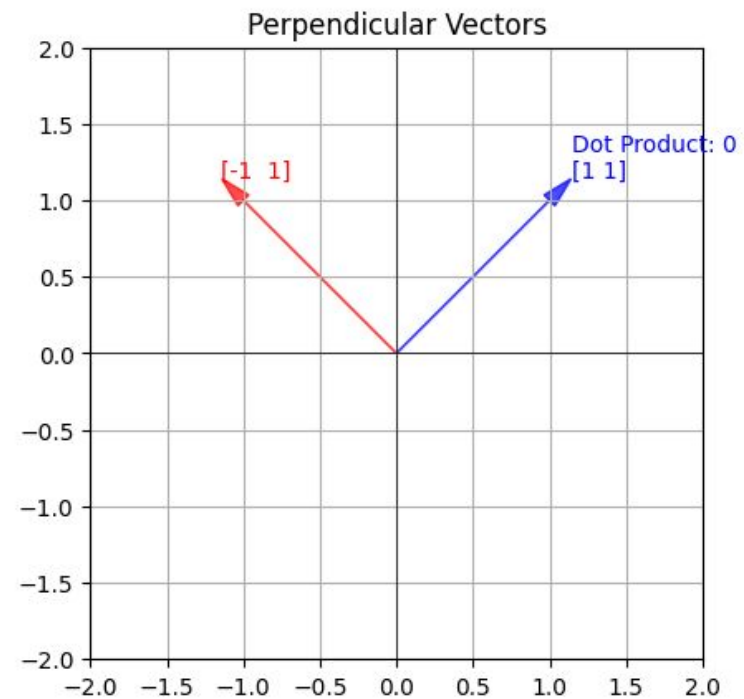
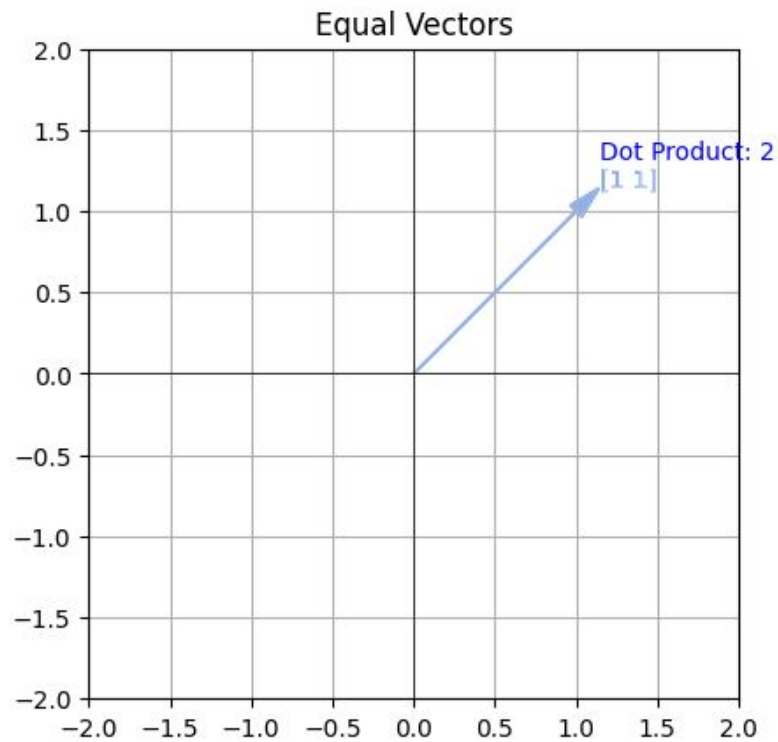
Vectors x and y:

x = [2 3 4]

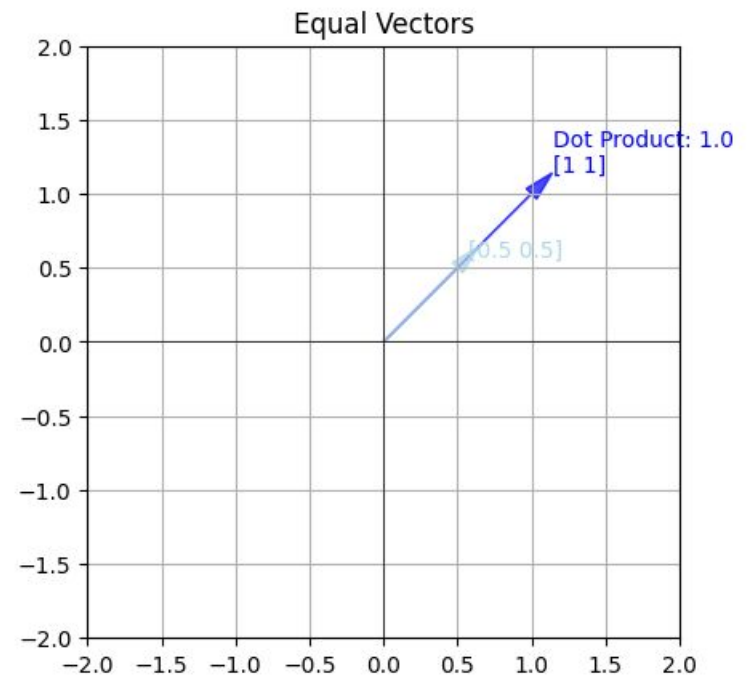
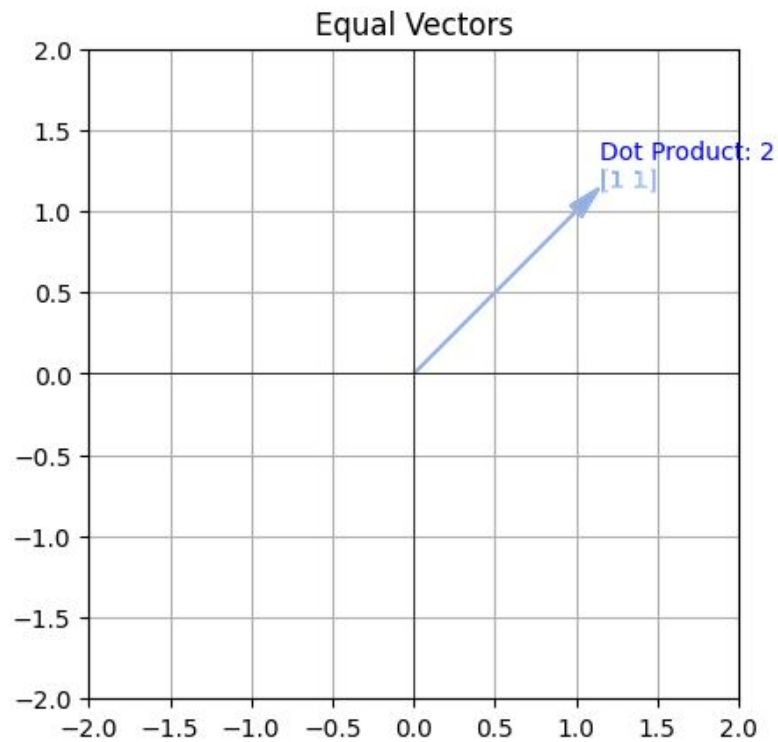
y = [ 1 0 -1]

Dot Product (x · y) = -2

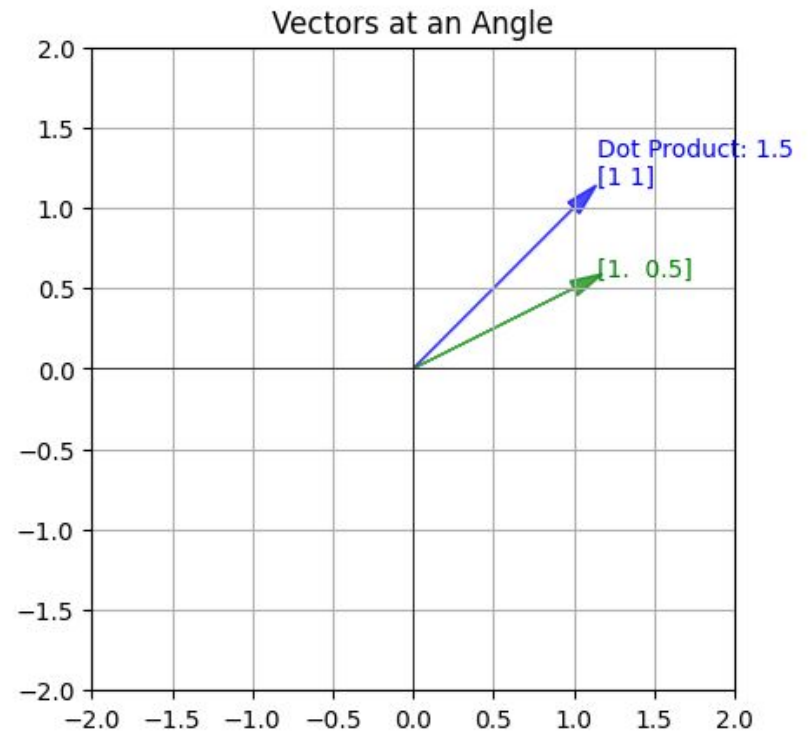
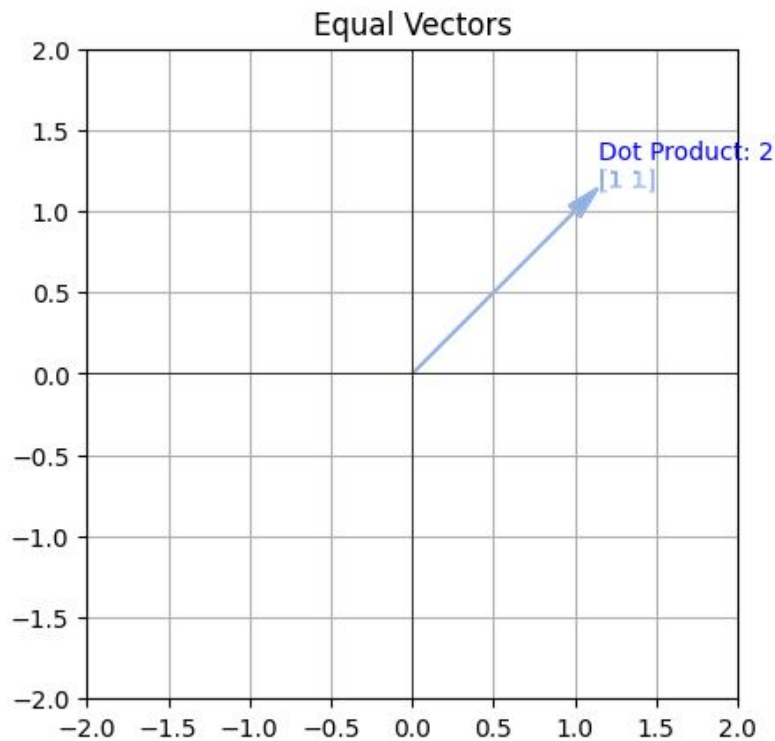
# Why do We need Dot Product?



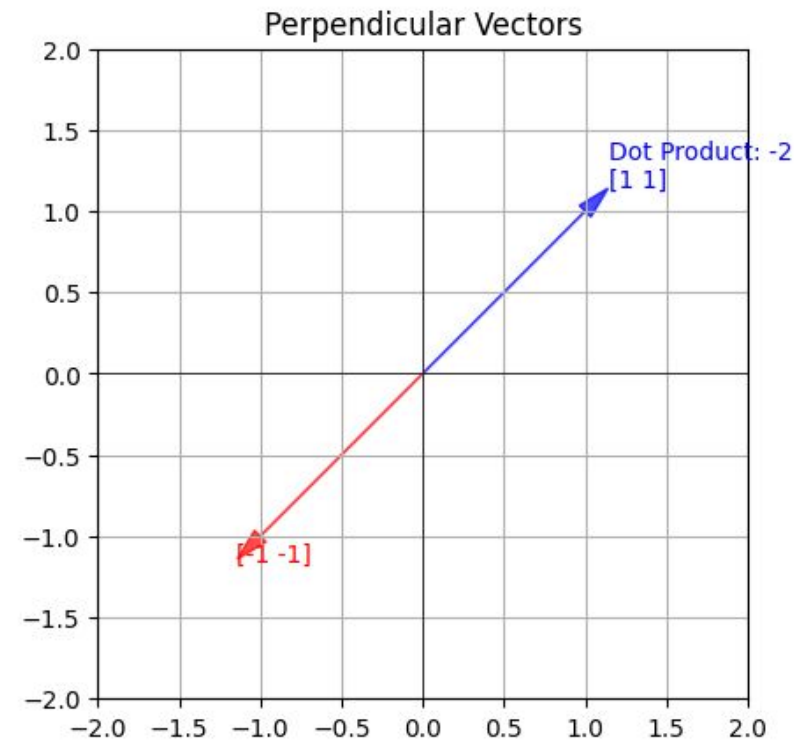
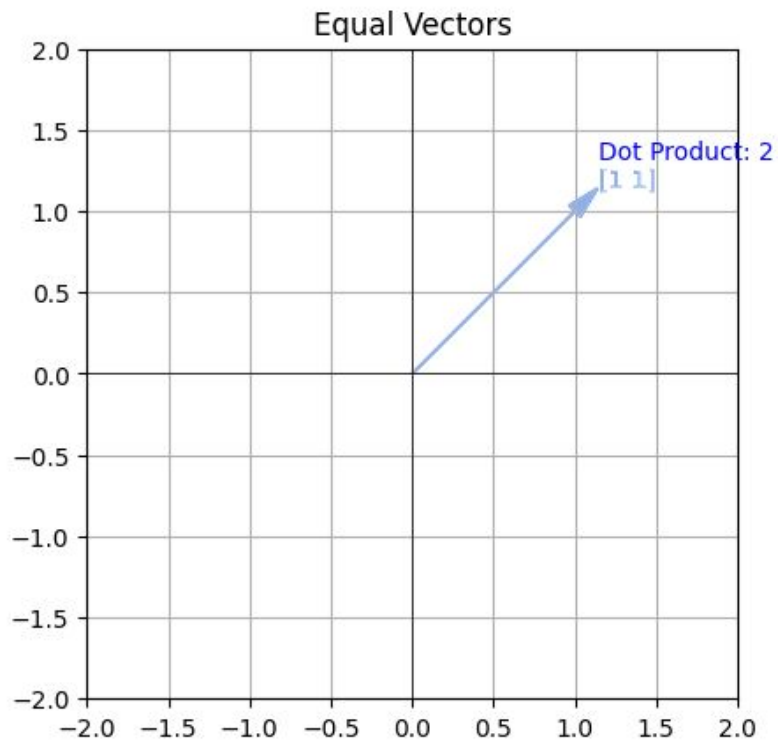
# Why do We need Dot Product?



# Why do We need Dot Product?



# Why do We need Dot Product?



# Why do We need Dot Product?

Dot product (similarity) between 'apple' and 'banana': 911.4376831054688  
Dot product (similarity) between 'apple' and 'lion': 161.2523651123047  
Embedding Shape: (300,)

```
1 import spacy
2 import numpy as np
3
4 # Load the medium English model in spaCy for better embeddings
5 nlp = spacy.load("en_core_web_md")
6
7 # Define the words
8 words = ["apple", "banana", "lion"]
9
10 # Get the embeddings for the words
11 embeddings = {word: nlp(word).vector for word in words}
12
13 # Display the embedding vectors
14 for word, vector in embeddings.items():
15     print(f"Embedding vector for '{word}':")
16     print(vector, "\n")
17
18 # Calculate the dot product between 'apple' and 'banana', and 'apple' and 'lion'
19 similarity_apple_banana = np.dot(embeddings['apple'], embeddings['banana'])
20 similarity_apple_lion = np.dot(embeddings['apple'], embeddings['lion'])
21
22 # Print the results
23 print(f"Dot product (similarity) between 'apple' and 'banana': {similarity_apple_banana}")
24 print(f"Dot product (similarity) between 'apple' and 'lion': {similarity_apple_lion}")
25
```

# Why do We need Dot Product?

```
Dot product (similarity) between 'apple' and 'banana': 39.10390090942383  
Dot product (similarity) between 'apple' and 'lion': 35.11163330078125  
Embedding Shape: (96,)
```

```
1  import spacy  
2  import numpy as np  
3  
4  # Load the medium English model in spaCy for better embeddings  
5  nlp = spacy.load("en_core_web_sm")  
6  
7  # Define the words  
8  words = ["apple", "banana", "lion"]  
9  
10 # Get the embeddings for the words  
11 embeddings = {word: nlp(word).vector for word in words}  
12  
13 # Display the embedding vectors  
14 for word, vector in embeddings.items():  
15     print(f"Embedding vector for '{word}':")  
16     print(vector, "\n")  
17  
18 # Calculate the dot product between 'apple' and 'banana', and 'apple' and 'lion'  
19 similarity_apple_banana = np.dot(embeddings['apple'], embeddings['banana'])  
20 similarity_apple_lion = np.dot(embeddings['apple'], embeddings['lion'])  
21  
22 # Print the results  
23 print(f"Dot product (similarity) between 'apple' and 'banana': {similarity_apple_banana}")  
24 print(f"Dot product (similarity) between 'apple' and 'lion': {similarity_apple_lion}")  
25 print(f"Embedding Shape: {embeddings['apple'].shape}")  
26
```



# Vectors & Numpy Arrays Vector Normalization



SPECIALIZATION

**CS316**  
**INTRODUCTION TO AI AND DATA  
SCIENCE**

**CHAPTER 3**  
**PANDAS AND NUMPY**

**LECTURE 1**

*PANDAS*

# Vector Normalization (L2)

- **Definition:**

- Vector normalization is the process of scaling a vector to have a unit length, i.e., a magnitude of 1. This is accomplished by dividing each component of the vector by its norm (magnitude).

- **Compact Math Form:**

- The normalized vector  $\hat{\mathbf{v}}$  of a vector  $\mathbf{v}$  is given by:

$$\hat{\mathbf{v}} = \frac{\mathbf{v}}{\|\mathbf{v}\|}$$

- **Expanded Math Form:**

- Given a vector  $\mathbf{v} = [v_1, v_2, \dots, v_n]$ , its normalized form  $\hat{\mathbf{v}}$  is:

$$\hat{\mathbf{v}} = \left[ \frac{v_1}{\sqrt{v_1^2 + v_2^2 + \dots + v_n^2}}, \frac{v_2}{\sqrt{v_1^2 + v_2^2 + \dots + v_n^2}}, \dots, \frac{v_n}{\sqrt{v_1^2 + v_2^2 + \dots + v_n^2}} \right]$$

- **Purpose:**

- **Similarity of Two Vectors:**

- Normalization is crucial for methods like cosine similarity, where the similarity between two vectors is measured based on the angle between them, rather than their magnitude. By normalizing vectors, we ensure that the similarity measure reflects directional similarity, independent of vector length.

# Vector Normalization – Example

- Expanded Math Form:

- For vector  $\mathbf{v} = [3, 4]$ :

$$\|\mathbf{v}\| = \sqrt{3^2 + 4^2} = 5$$

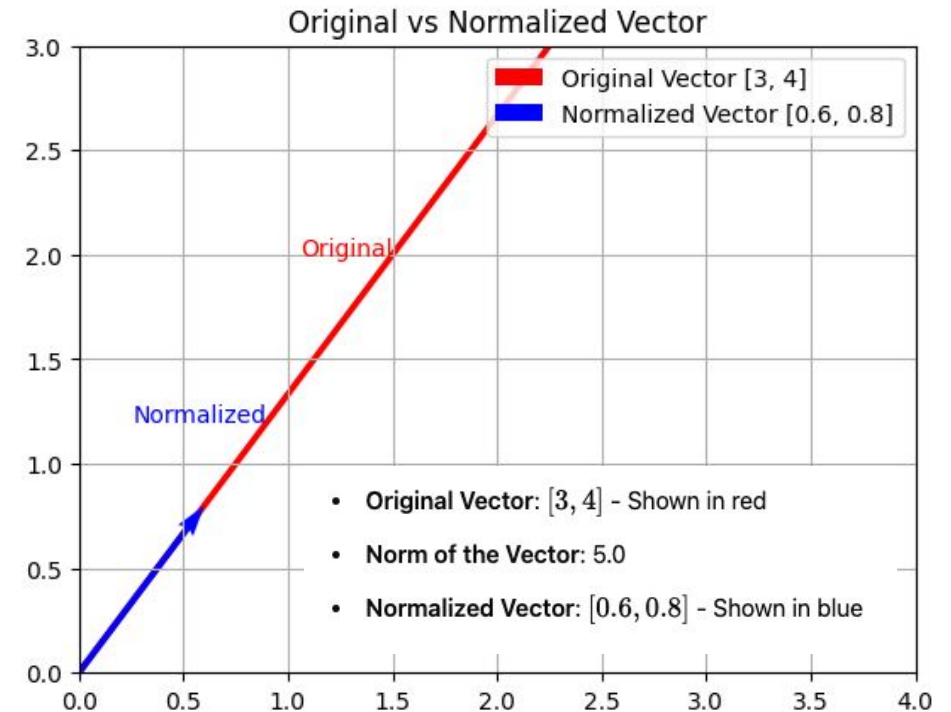
$$\hat{\mathbf{v}} = \left[ \frac{3}{5}, \frac{4}{5} \right]$$

- Numerical Example:

- Normalize  $\mathbf{v} = [3, 4]$ .
  - Norm  $\|\mathbf{v}\| = 5$ .
  - Normalized Vector  $\hat{\mathbf{v}} = [0.6, 0.8]$ .

- NumPy Implementation:

```
python Copy code  
  
import numpy as np  
v = np.array([3, 4])  
norm_v = np.linalg.norm(v)  
normalized_v = v / norm_v  
print("Original Vector:", v)  
print("Norm of v:", norm_v)  
print("Normalized Vector:", normalized_v)
```



# L1-Normalization

**L1 Norm (Manhattan Distance):**

- **Compact Form:**

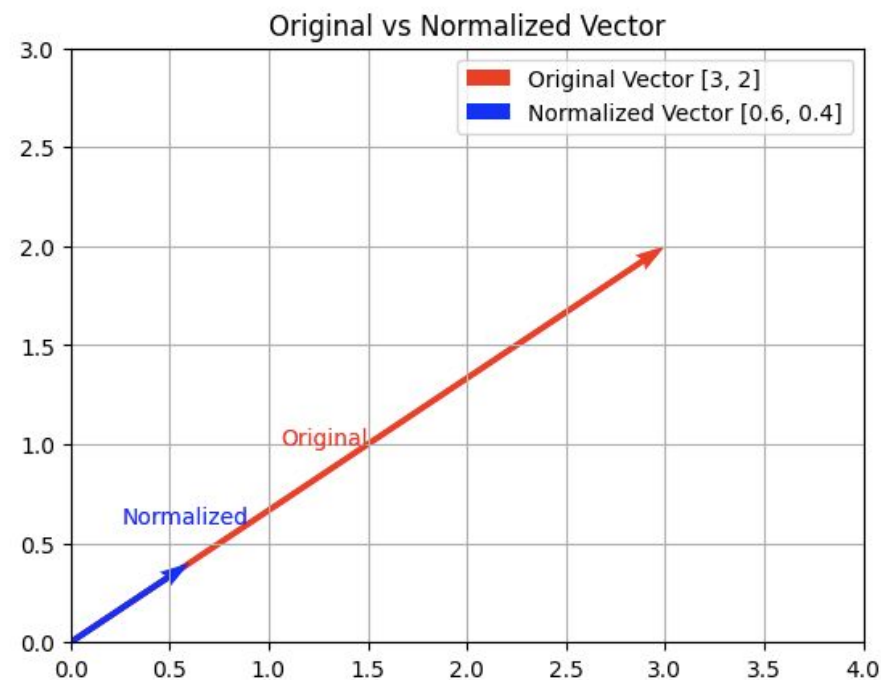
$$\|\mathbf{x}\|_1 = \sum_{i=1}^n |x_i|$$

- **Expanded Form:**

- For vector  $\mathbf{x} = [x_1, x_2, \dots, x_n]$ :

$$\|\mathbf{x}\|_1 = |x_1| + |x_2| + \dots + |x_n|$$

- **Purpose:** Measures the sum of the absolute values of the components. It is useful in scenarios where differences of any size are equally important.



# L2-Normalization

## L2 Norm (Euclidean Distance):

- **Compact Form:**

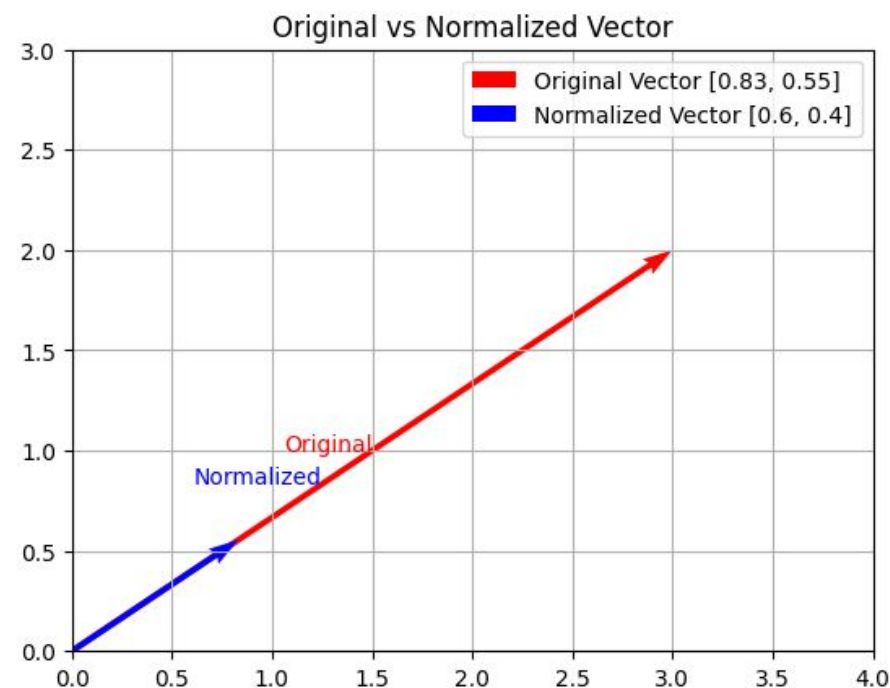
$$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^n x_i^2}$$

- **Expanded Form:**

- For vector  $\mathbf{x} = [x_1, x_2, \dots, x_n]$ :

$$\|\mathbf{x}\|_2 = \sqrt{x_1^2 + x_2^2 + \dots + x_n^2}$$

- **Purpose:** Measures the straight-line distance from the origin to the point in n-dimensional space, which is useful for determining actual distances between points.



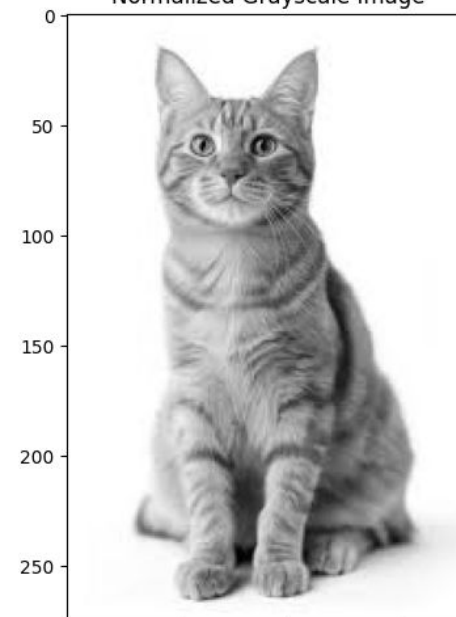
# Normalization in Computer Vision

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import requests
4 from PIL import Image
5 from io import BytesIO
6
7 # Load the image from a URL
8 image_url = "https://www.riotu-lab.org/cs313/cat.jpg"
9 response = requests.get(image_url)
10 img = Image.open(BytesIO(response.content))
11
12 # Convert the image to a numpy array
13 image = np.array(img)
14
15 # Convert the image to grayscale for simplicity
16 if image.ndim == 3:
17     grayscale_image = np.dot(image[... , :3], [0.2989, 0.5870, 0.1140])
18 else:
19     grayscale_image = image # if the image is already in grayscale
20
21 # Flatten the grayscale image to simulate a vector of pixel intensities
22 image_vector = grayscale_image.flatten()
23
24 # Compute the L2 norm of the image vector
25 l2_norm = np.linalg.norm(image_vector)
26
27 # Normalize the image vector
28 normalized_image_vector = image_vector / l2_norm
29
30 # Reshape normalized image back to the original grayscale image dimensions
31 normalized_image = normalized_image_vector.reshape(grayscale_image.shape)
32
33 # Plotting the original and normalized images
34 plt.figure(figsize=(12, 6))
35
36 # Plot original image
37 plt.subplot(1, 2, 1)
38 plt.imshow(grayscale_image, cmap='gray')
39 plt.title("Original Grayscale Image")
```

Original Grayscale Image



Normalized Grayscale Image



```
1 print(normalized_image_vector)
```

```
array([0.00524478, 0.00524478, 0.00524478, ..., 0.00524478, 0.00524478,
       0.00524478])
[0.00524478 0.00524478 0.00524478 ... 0.00524478 0.00524478 0.00524478]
```

```
[43] 1 print(image_vector)
```

```
[254.9745 254.9745 254.9745 ... 254.9745 254.9745 254.9745]
```

# Normalization in NLP

```
1 import spacy
2 import numpy as np
3
4 # Load the medium English model in spaCy for word embeddings
5 nlp = spacy.load("en_core_web_md")
6
7 # Define the words
8 words = ["apple", "banana", "lion"]
9
10 # Get the embeddings for the words
11 embeddings = {word: nlp(word).vector for word in words}
12
13 # Display the embedding vectors and calculate their norms (L2 norm)
14 for word, vector in embeddings.items():
15     # Compute the L2 norm (Euclidean norm) of the word embedding
16     norm = np.linalg.norm(vector)
17     print(f"Vector norm (L2) for '{word}': {norm}")
18
```

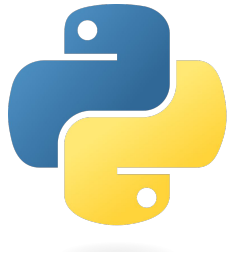
Vector norm (L2) for 'apple': 43.36647415161133  
Vector norm (L2) for 'banana': 31.6203556060791  
Vector norm (L2) for 'lion': 55.14573287963867

Vector norm (L2) for 'apple': 43.36647415161133  
Vector norm (L2) for 'banana': 31.6203556060791  
Vector norm (L2) for 'lion': 55.14573287963867

# Normalization in NLP

```
1 import spacy
2 import numpy as np
3
4 # Load the medium English model in spaCy for word embeddings
5 nlp = spacy.load("en_core_web_md")
6
7 # Define the words
8 words = ["apple", "banana", "lion"]
9
10 # Get the embeddings for the words
11 embeddings = {word: nlp(word).vector for word in words}
12
13 # Function to normalize a vector
14 def normalize_vector(vector):
15     norm = np.linalg.norm(vector) # Calculate the L2 norm
16     if norm == 0: # To prevent division by zero
17         return vector
18     return vector / norm
19
20 # Normalize the embeddings and show the result
21 for word, vector in embeddings.items():
22     normalized_vector = normalize_vector(vector)
23
24     # Show original and normalized vector norms
25     print(f"Original L2 norm for '{word}': {np.linalg.norm(vector)}")
26     print(f"Normalized L2 norm for '{word}': {np.linalg.norm(normalized_vector)}")
27     print(f"Normalized vector for '{word}':\n{normalized_vector}\n")
28
```

```
1.0
Original L2 norm for 'apple': 43.36647415161133
Normalized L2 norm for 'apple': 1.0
Normalized vector for 'apple':
[-0.02325298 -0.0468288 -0.0148006  0.06209405  0.00732617 -0.06148067
 -0.08617717  0.12616658 -0.11933412  0.01198114  0.12657934 -0.04823542
 -0.05749142  0.05338686  0.00906207 -0.05597181  0.07896883 -0.06115323
  0.05843454 -0.11089443 -0.04526769  0.11309428 -0.05865822 -0.03650285
 -0.08719639 -0.05750064 -0.05594414  0.08798733  0.01064832  0.02833064
 -0.00483484 -0.03240522 -0.0555429 -0.05133689  0.0328249 -0.04312087
  0.05970511  0.07801418  0.06552527 -0.03466042  0.02447513  0.03105394
  0.01761545  0.02147581  0.04973427  0.05345143  0.03137677 -0.00476566
  0.06601644  0.01765949  0.0379579  0.01013041  0.0476451 -0.05680425
 -0.06277199  0.0271869  0.00095576  0.05129308  0.04127843 -0.01273749
  0.06190266  0.08541851 -0.01359553 -0.00046061 -0.03268423 -0.0099736
 -0.04702942 -0.12144865  0.01810177  0.00215099 -0.01777687  0.04564355
 -0.02931758 -0.01038959  0.05871356  0.0341093 -0.10125333 -0.06029081
  0.05877351 -0.03483567  0.00474906  0.0393207 -0.00729227 -0.01800561
  0.02562809  0.10181367  0.12546559  0.10126255 -0.06089958  0.02357812
 -0.00602862  0.07747921  0.06238921 -0.04965356  0.0086131 -0.00793977
  0.11659697 -0.06637616 -0.0374921 -0.07797268 -0.00067771 -0.03695481]
```



# Vectors & Numpy Arrays Cosine Similarity



**CS316**  
**INTRODUCTION TO AI AND DATA  
SCIENCE**

**CHAPTER 3**  
**PANDAS AND NUMPY**

**LECTURE 1**

*PANDAS*

# Cosine Similarity of Two Vectors

- **Definition:**

- Cosine similarity is a measure of similarity between two non-zero vectors within an inner product space that measures the cosine of the angle between them.

- **Math Equation:**

- **Compact Form:**

$$\text{Cosine Similarity}(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\| \|\mathbf{y}\|}$$

- **Expanded Form:**

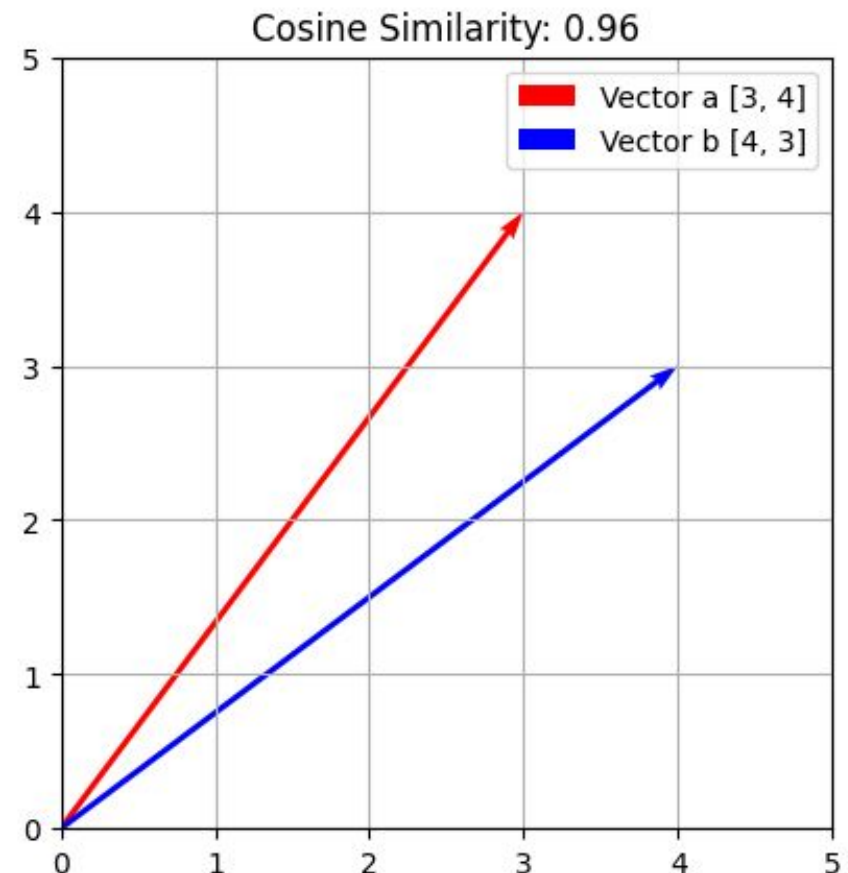
- Given vectors  $\mathbf{x} = [x_1, x_2, \dots, x_n]$  and  $\mathbf{y} = [y_1, y_2, \dots, y_n]$ :

$$\text{Cosine Similarity} = \frac{x_1 y_1 + x_2 y_2 + \dots + x_n y_n}{\sqrt{x_1^2 + x_2^2 + \dots + x_n^2} \cdot \sqrt{y_1^2 + y_2^2 + \dots + y_n^2}}$$

- **Purpose:**

- **Similarity Measurement:**

- Cosine similarity is widely used in various applications such as text analysis, where it helps in identifying the similarity between documents. It is especially useful in high-dimensional positive spaces like text data in TF-IDF representation.
- In machine learning, it helps in clustering and classification by measuring how similar the data objects are, irrespective of their magnitude.



# Cosine Similarity of Two Vectors

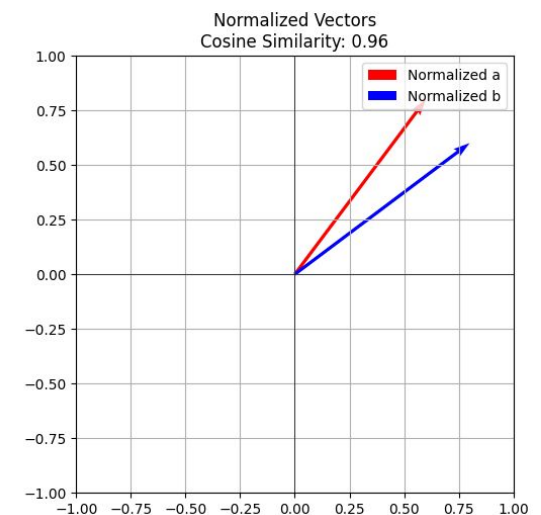
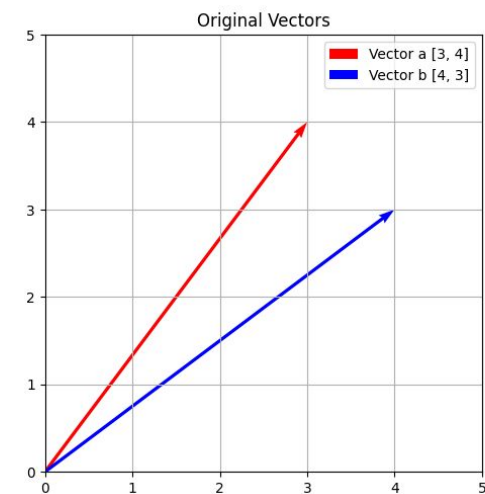
```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 # Define the vectors
5 a = np.array([3, 4])
6 b = np.array([4, 3])
7
8 # Calculate the dot product
9 dot_product = np.dot(a, b)
10
11 # Calculate the norms of the vectors
12 norm_a = np.linalg.norm(a)
13 norm_b = np.linalg.norm(b)
14
15 # Calculate the cosine similarity
16 cosine_similarity = dot_product / (norm_a * norm_b)
17
18 print("Cosine Similarity:", cosine_similarity)
19
```

**Numerical Example:**

Consider two vectors in a 2D space:

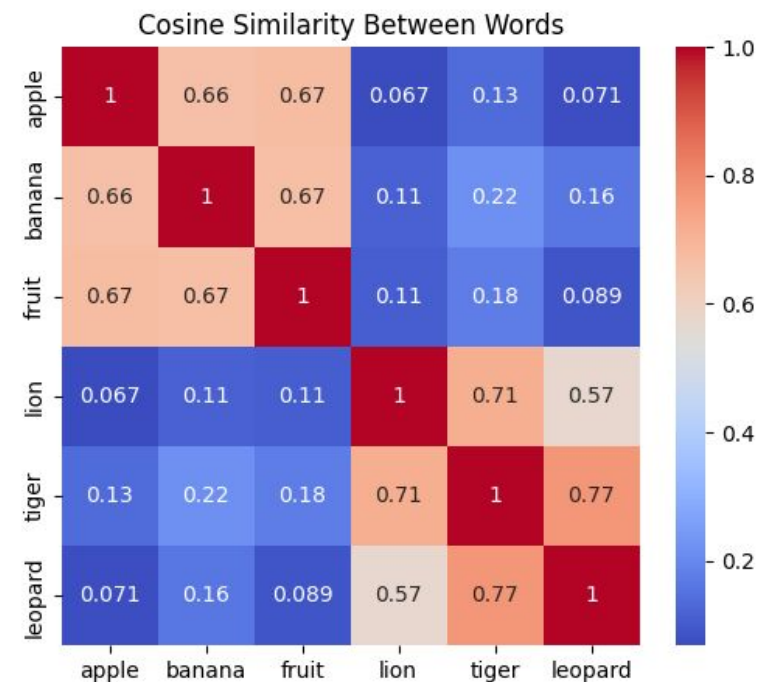
- $\mathbf{a} = [3, 4]$
- $\mathbf{b} = [4, 3]$

$$\text{Cosine Similarity}(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\| \|\mathbf{y}\|}$$



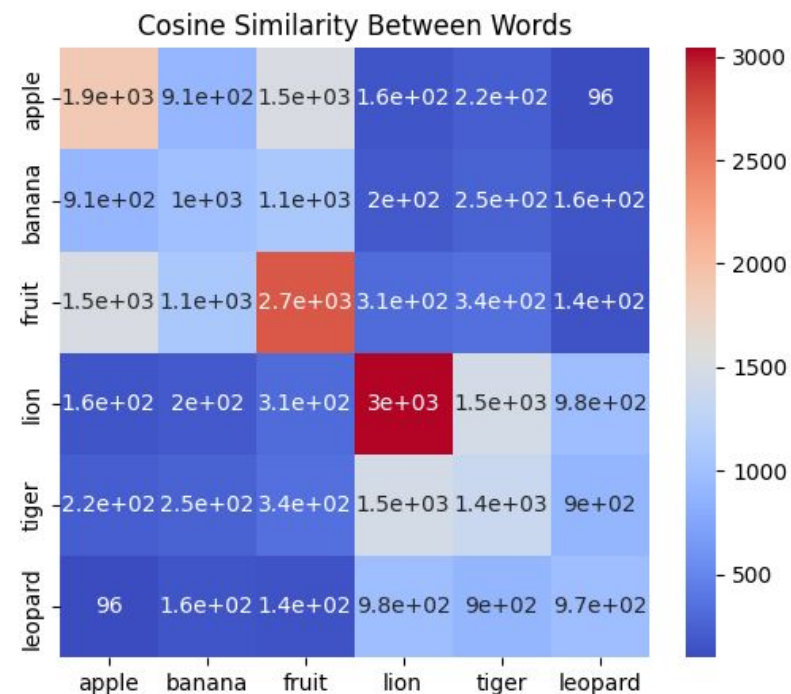
# Cosine Similarity of Two Vectors

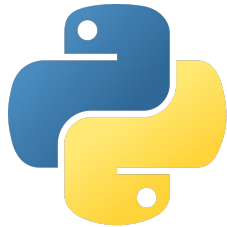
```
1 import spacy
2 import numpy as np
3 import matplotlib.pyplot as plt
4 import seaborn as sns
5
6 # Load the medium English model in spaCy for word embeddings
7 nlp = spacy.load("en_core_web_lg")
8
9 # Define the words
10 words = ["apple", "banana", "fruit", "lion", "tiger", "leopard"]
11
12 # Get the embeddings for the words
13 embeddings = {word: nlp(word).vector for word in words}
14
15 # Function to normalize a vector
16 def normalize_vector(vector):
17     norm = np.linalg.norm(vector) # Calculate the L2 norm
18     if norm == 0: # To prevent division by zero
19         return vector
20     return vector / norm
21
22 # Normalize the embeddings
23 normalized_embeddings = {word: normalize_vector(vector) for word, vector in embeddings.items()}
24
25 # Calculate pairwise cosine similarities
26 cosine_similarities = np.zeros((len(words), len(words)))
27
28 for i, word1 in enumerate(words):
29     for j, word2 in enumerate(words):
30         cosine_similarities[i, j] = np.dot(normalized_embeddings[word1], normalized_embeddings[word2])
31
32 # Plotting the cosine similarities using a heatmap
33 plt.figure(figsize=(6, 5))
34 sns.heatmap(cosine_similarities, annot=True, cmap='coolwarm', xticklabels=words, yticklabels=words)
35 plt.title('Cosine Similarity Between Words')
36 plt.show()
37
```



# The Impact of Normalization

```
1 import spacy
2 import numpy as np
3 import matplotlib.pyplot as plt
4 import seaborn as sns
5
6 # Load the medium English model in spaCy for word embeddings
7 nlp = spacy.load("en_core_web_lg")
8
9 # Define the words
10 words = ["apple", "banana", "fruit", "lion", "tiger", "leopard"]
11
12 # Get the embeddings for the words
13 embeddings = {word: nlp(word).vector for word in words}
14
15 # Function to normalize a vector
16 def normalize_vector(vector):
17     norm = np.linalg.norm(vector) # Calculate the L2 norm
18     if norm == 0: # To prevent division by zero
19         return vector
20     return vector / norm
21
22 # Normalize the embeddings
23 normalized_embeddings = {word: normalize_vector(vector) for word, vector in embeddings.items()}
24
25 # Calculate pairwise cosine similarities
26 similarities = np.zeros((len(words), len(words)))
27
28 for i, word1 in enumerate(words):
29     for j, word2 in enumerate(words):
30         similarities[i, j] = np.dot(embeddings[word1], embeddings[word2])
31
32 # Plotting the cosine similarities using a heatmap
33 plt.figure(figsize=(6, 5))
34 sns.heatmap(similarities, annot=True, cmap='coolwarm', xticklabels=words, yticklabels=words)
35 plt.title('Cosine Similarity Between Words')
36 plt.show()
```





# Matrices

## 4.1 Understanding Matrices



SPECIALIZATION

**CS316**  
**INTRODUCTION TO AI AND DATA  
SCIENCE**

**CHAPTER 3**  
**PANDAS AND NUMPY**

**LECTURE 1**

*PANDAS*

# 4.1 Understanding Matrices

- Linear Algebra Concept:

- Definition of Matrices:

- Matrices are two-dimensional arrays of numbers, denoted as  $\mathbf{A}$  with elements  $a_{ij}$  where  $i$  and  $j$  are row and column indices, respectively.

- Applications in Transformations:

- Matrices can represent linear transformations in space, such as rotations, scaling, and translations.

- Compact Form:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

- Expanded Form:

- For a matrix  $\mathbf{A}$  with general elements:

$$\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

## NumPy Implementation:

- Creating Matrices with 2D Arrays:

```
python Copy code

import numpy as np
A = np.array([[1, 2], [3, 4]])
```

- Exploring Matrix Attributes:

- Shape:** Determines the dimensions of the matrix.
- Dtype:** Indicates the data type of matrix elements.

```
python Copy code

print("Shape of A:", A.shape)
print("Data type of A:", A.dtype)
```

## 4.2 Matrix Operations

- **Matrix Addition and Subtraction:**

- Addition or subtraction of two matrices of the same dimension results in a matrix where each element is the sum or difference of corresponding elements.

- **Compact Form:**

$$\mathbf{A} + \mathbf{B} = \begin{bmatrix} a_{11} + b_{11} & a_{12} + b_{12} \\ a_{21} + b_{21} & a_{22} + b_{22} \end{bmatrix}$$

- **Scalar Multiplication:**

- Multiplying a matrix by a scalar multiplies each element of the matrix by the scalar.

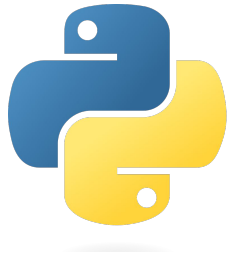
- **Compact Form:**

$$c\mathbf{A} = \begin{bmatrix} c \cdot a_{11} & c \cdot a_{12} \\ c \cdot a_{21} & c \cdot a_{22} \end{bmatrix}$$

### Performing Matrix Operations (+, -, \*):

python

```
import numpy as np
A = np.array([[1, 2], [3, 4]])
B = np.array([[2, 0], [1, 3]])
addition = A + B
subtraction = A - B
scalar_multiplication = 2 * A
```



# Matrices

## 4.2 Matrix Operation



**CS316**  
**INTRODUCTION TO AI AND DATA  
SCIENCE**

**CHAPTER 3**  
**PANDAS AND NUMPY**

**LECTURE 1**  
*PANDAS*

## 4.2 Matrix Operations

```
1 import numpy as np
2
3 # Define the matrices
4 A = np.array([[1, 2], [3, 4]])
5 B = np.array([[2, 0], [1, 3]])
6
7 # Perform operations
8 addition = A + B
9 subtraction = A - B
10 scalar_multiplication = 2 * A
11
12 # Print the results
13 print("Matrix A:\n", A)
14 print("\nMatrix B:\n", B)
15 print("\nAddition of A and B:\n", addition)
16 print("\nSubtraction of A from B:\n", subtraction)
17 print("\nScalar Multiplication of A by 2:\n", scalar_multiplication)
18
```

Matrix A:

```
[[1 2]
 [3 4]]
```

Matrix B:

```
[[2 0]
 [1 3]]
```

Addition of A and B:

```
[[3 2]
 [4 7]]
```

Subtraction of A from B:

```
[[ -1  2]
 [ 2  1]]
```

Scalar Multiplication of A by 2:

```
[[2 4]
 [6 8]]
```

## 4.2 Matrix Operations

### Matrix Multiplication:


- The product of two matrices is a new matrix where each element is computed as the dot product of rows of the first matrix with columns of the second.

- **Compact Form:**

$$\mathbf{A} \times \mathbf{B} = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}$$

### Matrix Multiplication (np.matmul, @):

python

 Copy code

```
matrix_multiplication = np.matmul(A, B)
# or using the @ operator
matrix_multiplication_alt = A @ B
```

- **Given Matrices:**

- Matrix **A**:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

- Matrix **B**:

$$\begin{bmatrix} 2 & 0 \\ 1 & 3 \end{bmatrix}$$

- **Matrix Multiplication:**

- Matrix multiplication involves the dot product of rows of **A** with columns of **B**.

- **Calculation:**

$$\mathbf{C} = \mathbf{A} \times \mathbf{B} = \begin{bmatrix} (1 \cdot 2 + 2 \cdot 1) & (1 \cdot 0 + 2 \cdot 3) \\ (3 \cdot 2 + 4 \cdot 1) & (3 \cdot 0 + 4 \cdot 3) \end{bmatrix} = \begin{bmatrix} 4 & 6 \\ 10 & 12 \end{bmatrix}$$

# Dot Product of Two Matrices

- **Definition:**

- The dot product of two matrices involves element-wise multiplication of two matrices of the same size. The result is another matrix where each element is the product of elements at corresponding positions in the original matrices.

- **Math Equation:**

- **Compact Form:**

$$\mathbf{C} = \mathbf{A} \odot \mathbf{B}$$

- **Example:**


- For matrices  $\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$  and  $\mathbf{B} = \begin{bmatrix} 2 & 0 \\ 1 & 3 \end{bmatrix}$ :

$$\mathbf{C} = \mathbf{A} \odot \mathbf{B} = \begin{bmatrix} 1 \cdot 2 & 2 \cdot 0 \\ 3 \cdot 1 & 4 \cdot 3 \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 3 & 12 \end{bmatrix}$$

- **Purpose:**

- This operation is often used in component-wise calculations required in applied mathematics and certain types of statistical analyses.

python

 Copy code

```
import numpy as np

# Define the matrices
A = np.array([[1, 2], [3, 4]])
B = np.array([[2, 0], [1, 3]])

# Compute the dot product
C = A * B # Element-wise multiplication

print("Matrix A:\n", A)
print("Matrix B:\n", B)
print("Dot Product (Element-wise multiplication) of A and B:\n", C)
```

# Dot Product of Two Matrices

```
1 import numpy as np
2
3 # Define the matrices
4 A = np.array([[1, 2], [3, 4]])
5 B = np.array([[2, 0], [1, 3]])
6
7 # Compute the dot product of matrices A and B
8 dot_product = np.dot(A, B)
9
10 # Print the results
11 print("Matrix A:\n", A)
12 print("\nMatrix B:\n", B)
13 print("\nDot Product of A and B:\n", dot_product)
14
```

Matrix A:

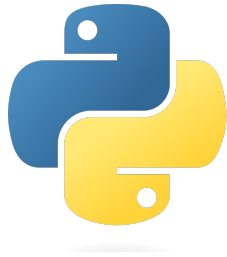
```
[[1 2]
 [3 4]]
```

Matrix B:

```
[[2 0]
 [1 3]]
```

Dot Product of A and B:

```
[[ 4  6]
 [10 12]]
```



# Matrices

## 4.3 Special Matrices



**CS316**  
**INTRODUCTION TO AI AND DATA  
SCIENCE**

**CHAPTER 3**  
**PANDAS AND NUMPY**

**LECTURE 1**  
*PANDAS*

# Identity Matrix

Identity Matrix  $\mathbf{I}_3$ :

$$\mathbf{I}_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- **Definition:** An identity matrix  $\mathbf{I}$  is a square matrix in which all the elements of the principal diagonal are ones, and all other elements are zeros.

- **Compact Form:**

$$\mathbf{I}_n = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}$$

- **Properties:**

- Multiplicative Identity:  $\mathbf{A} \times \mathbf{I} = \mathbf{A}$

```
1 import numpy as np
2 # Create a 4x4 identity matrix
3 I = np.eye(4) # 4x4 identity matrix
4 print("\nIdentity Matrix\n", I)
```

Identity Matrix  
[[1. 0. 0. 0.]  
[0. 1. 0. 0.]  
[0. 0. 1. 0.]  
[0. 0. 0. 1.]]

# Diagonal Matrix

- **Definition:** A diagonal matrix  $\mathbf{D}$  is a matrix in which the entries outside the main diagonal are all zero.
- **Compact Form:**

$$\mathbf{D} = \begin{bmatrix} d_1 & 0 & \cdots & 0 \\ 0 & d_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & d_n \end{bmatrix}$$

Diagonal Matrix  $\mathbf{D}_4$ :

$$\mathbf{D}_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 4 \end{bmatrix}$$

```
1 # Create a diagonal matrix
2 D = np.diag([1, 2, 3, 4])
3 # Extract the diagonal elements from D
4 diagonal_elements = np.diag(D)
5 print("\nDiagonal Matrix\n", D)
6 print("\nDiagonal Elements\n", diagonal_elements)
```

Diagonal Matrix

```
[[1 0 0 0]
 [0 2 0 0]
 [0 0 3 0]
 [0 0 0 4]]
```

Diagonal Elements

```
[1 2 3 4]
```



# Module 5: Systems of Linear Equations

## 5.1 Solving Linear Systems



SPECIALIZATION

**CS316**  
**INTRODUCTION TO AI AND DATA  
SCIENCE**

**CHAPTER 3**  
**PANDAS AND NUMPY**

**LECTURE 1**

*PANDAS*



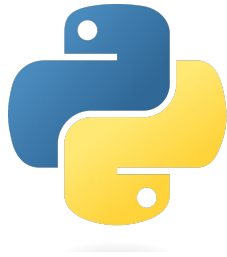
# CS316 INTRODUCTION TO AI AND DATA SCIENCE

## CHAPTER 4 Integrating Linear Algebra with NumPy: Practical Foundations for Data Science

### LECTURE 2 *Determinant, Rank, Full Rank Matrix, Inverse Matrix, Pseudo-Inverse Matrix, Solving Linear Systems*

Prof. Anis Koubaa

SEP 2024



# Module 5: Determinants and Rank

## 5.1 Determinants



## CS316 INTRODUCTION TO AI AND DATA SCIENCE

### CHAPTER 3 PANDAS AND NUMPY

#### LECTURE 1

#### *PANDAS*

# 6.1 Determinants

- **Definition of Determinants:**

- The determinant of a square matrix  $\mathbf{A}$  is a scalar value that is computed from the elements of the matrix. It is defined recursively and can be calculated via Laplace expansion or by using elementary row operations.

- **Mathematical Formula (Compact):**

- For a 2x2 matrix  $\mathbf{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$ :

$$\det(\mathbf{A}) = ad - bc$$

- **Expanded Formula for a 3x3 Matrix:**

- For matrix  $\mathbf{A} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$ :

$$\det(\mathbf{A}) = a(ei - fh) - b(di - fg) + c(dh - eg)$$

- **Numerical Example:**

- **Problem Statement:**

- Calculate the determinant of the matrix  $\mathbf{A}$  and interpret the result.

- **Given Matrix  $\mathbf{A}$ :**

$$\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

- **Solution:**

- Using the compact formula:  $\det(\mathbf{A}) = 1(4) - 2(3) = 4 - 6 = -2$
- Interpretation: Since  $\det(\mathbf{A}) \neq 0$ , matrix  $\mathbf{A}$  is invertible, and the transformation it represents is area changing by a factor of 2 and reversing the orientation (indicated by the negative sign).

- **Significance of Determinants:**

- **Invertibility:** A non-zero determinant indicates that the matrix is invertible.
- **Volume and Orientation:** The absolute value of the determinant represents the volume scaling factor of the transformation defined by the matrix, and its sign indicates the orientation (positive or negative).

# 6.1 Determinants

- **Definition of Determinants:**

- The determinant of a square matrix  $\mathbf{A}$  is a scalar value that is computed from the elements of the matrix. It is defined recursively and can be calculated via Laplace expansion or by using elementary row operations.

- **Mathematical Formula (Compact):**

- For a 2x2 matrix  $\mathbf{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$ :

$$\det(\mathbf{A}) = ad - bc$$

- **Expanded Formula for a 3x3 Matrix:**

- For matrix  $\mathbf{A} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$ :


$$\det(\mathbf{A}) = a(ei - fh) - b(di - fg) + c(dh - eg)$$

- **Calculating Determinants:**

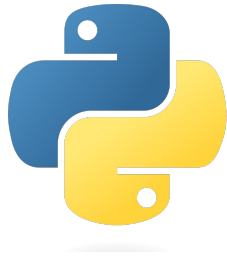
- The determinant can be computed using the `np.linalg.det()` function in Python's NumPy library.

- **Python Code Example:**

python

 Copy code

```
import numpy as np
A = np.array([[1, 2], [3, 4]])
det_A = np.linalg.det(A)
print("Determinant of Matrix A:", det_A)
```



## Module 5: Determinants and Rank

### 5.2 Rank of a Matrix



## CS316 INTRODUCTION TO AI AND DATA SCIENCE

### CHAPTER 3 PANDAS AND NUMPY

#### LECTURE 1

#### *PANDAS*

## 6.2 Rank of a Matrix

- **Definition of Rank:**
  - The rank of a matrix  $\mathbf{A}$  is the dimension of the vector space generated (spanned) by its columns. This also corresponds to the maximum number of linearly independent columns or rows in  $\mathbf{A}$ .

- **Mathematical Description:**

- **Compact Form:**

$$\text{Rank}(\mathbf{A}) = \dim(\text{col}(\mathbf{A}))$$

- **Expanded Form:**

$$\text{Rank}(\mathbf{A}) = \text{number of pivot positions in } \mathbf{A} = \dim(\text{row}(\mathbf{A}))$$

## 6.2 Rank of a Matrix

- **Importance of Rank:**
  - **System Solutions:** Knowing the rank of  $\mathbf{A}$  is crucial in determining the solvability of the system  $\mathbf{Ax} = \mathbf{b}$ . If  $\mathbf{A}$  is of full column rank, the system has a unique solution when  $\mathbf{b}$  is in the column space of  $\mathbf{A}$ .
  - **Dimensionality and Data Insights:** In data science, the rank of the feature matrix affects model complexity and generalization. A lower rank might indicate redundancy or correlations among features, suggesting potential for dimensionality reduction.

## 6.2 Rank of a Matrix

- Numerical Example:

- Problem Statement:

- Calculate the rank of matrix **A** and discuss its implications.

- Given Matrix **A**:

$$\mathbf{A} = \begin{bmatrix} 2 & 4 & 1 \\ 8 & 16 & 2 \\ 0 & 0 & 1 \end{bmatrix}$$

- Solution:

- Calculation reveals that the first two columns of **A** are linearly dependent (second column is twice the first), and the third column is linearly independent.
- Rank** = 2 (since two columns are linearly independent).
- Implication:** The system  $\mathbf{Ax} = \mathbf{b}$  will have solutions depending on the vector **b**. If **b** is in the span of the columns of **A**, there will be infinitely many solutions; otherwise, no solution.

### NumPy Implementation:

- Determining Rank:

- Use `np.linalg.matrix_rank()` to compute the rank of a matrix.

- Python Code Example:

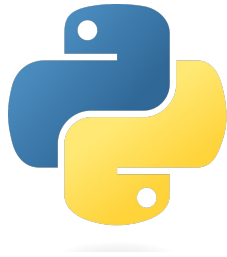
python

Copy code

```
import numpy as np
A = np.array([[2, 4, 1], [8, 16, 2], [0, 0, 1]])
rank_A = np.linalg.matrix_rank(A)
print("Rank of Matrix A:", rank_A)
```

```
1 import numpy as np
2 A = np.array([[2, 4, 1],
3               [8, 16, 2],
4               [0, 0, 1]])
5 rank_A = np.linalg.matrix_rank(A)
6 print("Rank of Matrix A:", rank_A)
7
```

Rank of Matrix A: 2



# Module 5: Systems of Linear Equations

## 5.3 Inverse Matrices



**CS316**  
**INTRODUCTION TO AI AND DATA  
SCIENCE**

**CHAPTER 3**  
**PANDAS AND NUMPY**

**LECTURE 1**  
*PANDAS*

# Inverse Matrices

- 5.2 Inverse Matrices

- Linear Algebra Concept:

- Definition of Inverse Matrices:

- An inverse matrix of a square matrix  $\mathbf{A}$  is a matrix  $\mathbf{A}^{-1}$ , which when multiplied by  $\mathbf{A}$  results in the identity matrix  $\mathbf{I}$ .

- Equation:

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

- Properties:

- $\mathbf{A}$  must be square and have a non-zero determinant.
- If  $\mathbf{A}$  is non-singular (i.e.,  $\det(\mathbf{A}) \neq 0$ ), then  $\mathbf{A}^{-1}$  exists.

## Numerical Application:

- Example:

- Given  $\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$  and  $\mathbf{b} = \begin{bmatrix} 5 \\ 11 \end{bmatrix}$ :

- Calculate  $\mathbf{A}^{-1}$  and solve for  $\mathbf{x}$ :

$$\mathbf{x} = \begin{bmatrix} -1.5 \\ 3.5 \end{bmatrix}$$

- Confirm solution:

$$\mathbf{A}\mathbf{x} = \mathbf{b}$$

- Calculating Inverses:

- Use `np.linalg.inv()` to compute the inverse of a matrix.

- Code:

```
python Copy code

import numpy as np
A = np.array([[1, 2], [3, 4]])
A_inv = np.linalg.inv(A)
```

- Using Inverses to Solve Systems:

- Solve  $\mathbf{A}\mathbf{x} = \mathbf{b}$  by computing  $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$ .

- Code:

```
python Copy code

b = np.array([5, 11])
x = np.dot(A_inv, b)
```

# Full Rank Matrix

- **Definition of Full Rank Matrix:**

- A matrix  $\mathbf{A}$  of dimensions  $m \times n$  is said to have **full rank** if the rank of  $\mathbf{A}$  is equal to the minimum of its number of rows  $m$  and columns  $n$ . Mathematically, this is expressed as:

$$\text{Rank}(\mathbf{A}) = \min(m, n)$$

- **Properties:**

- **Full Row Rank:** A matrix has full row rank if every row is linearly independent, which for  $m \leq n$ , means:

$$\text{Rank}(\mathbf{A}) = m$$

- **Full Column Rank:** A matrix has full column rank if every column is linearly independent, applicable when  $n \leq m$ , signified by:

$$\text{Rank}(\mathbf{A}) = n$$

- **Invertibility:** A square matrix  $m = n$  with full rank is invertible, which implies:

$$\text{Rank}(\mathbf{A}) = n = m$$

and  $\det(\mathbf{A}) \neq 0$ , indicating a non-zero determinant.

## Example 1: A 2x2 matrix with full rank

Consider the matrix:

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

- The matrix has 2 rows and 2 columns.
- To check if it's full rank, we need to see if the rows or columns are linearly independent.
- The rank of this matrix is 2 (since both rows are linearly independent).

Thus, **matrix A is full rank**, as its rank equals the smaller dimension (in this case, 2).

# Full Rank Matrix

- **Definition of Full Rank Matrix:**

- A matrix  $\mathbf{A}$  of dimensions  $m \times n$  is said to have **full rank** if the rank of  $\mathbf{A}$  is equal to the minimum of its number of rows  $m$  and columns  $n$ . Mathematically, this is expressed as:

$$\text{Rank}(\mathbf{A}) = \min(m, n)$$

- **Properties:**

- **Full Row Rank:** A matrix has full row rank if every row is linearly independent, which for  $m \leq n$ , means:

$$\text{Rank}(\mathbf{A}) = m$$

- **Full Column Rank:** A matrix has full column rank if every column is linearly independent, applicable when  $n \leq m$ , signified by:

$$\text{Rank}(\mathbf{A}) = n$$

- **Invertibility:** A square matrix  $m = n$  with full rank is invertible, which implies:

$$\text{Rank}(\mathbf{A}) = n = m$$

and  $\det(\mathbf{A}) \neq 0$ , indicating a non-zero determinant.

## Example 2: A 2x2 matrix that is not full rank

Consider the matrix:

$$B = \begin{pmatrix} 1 & 2 \\ 2 & 4 \end{pmatrix}$$

- This matrix also has 2 rows and 2 columns.
- But the second row is just 2 times the first row, meaning they are **linearly dependent**.
- The rank of this matrix is 1 (since we have only one linearly independent row or column).

Thus, **matrix B is not full rank** because its rank (1) is less than the smaller dimension of the matrix (2).

# Full Rank Matrix

- **Definition of Full Rank Matrix:**

- A matrix  $\mathbf{A}$  of dimensions  $m \times n$  is said to have **full rank** if the rank of  $\mathbf{A}$  is equal to the minimum of its number of rows  $m$  and columns  $n$ . Mathematically, this is expressed as:

$$\text{Rank}(\mathbf{A}) = \min(m, n)$$

- **Properties:**

- **Full Row Rank:** A matrix has full row rank if every row is linearly independent, which for  $m \leq n$ , means:

$$\text{Rank}(\mathbf{A}) = m$$

- **Full Column Rank:** A matrix has full column rank if every column is linearly independent, applicable when  $n \leq m$ , signified by:

$$\text{Rank}(\mathbf{A}) = n$$

- **Invertibility:** A square matrix  $m = n$  with full rank is invertible, which implies:

$$\text{Rank}(\mathbf{A}) = n = m$$

and  $\det(\mathbf{A}) \neq 0$ , indicating a non-zero determinant.

## Example 3: A 3x2 matrix with full rank

Consider the matrix:

$$C = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$$

- This matrix has 3 rows and 2 columns.
- Since it's a rectangular matrix, we check if the rows are independent.
- The rank of this matrix is 2 (since the two columns are linearly independent).

Thus, **matrix C** is **full rank** because its rank (2) equals the smaller dimension, which is the number of columns (2).

# Full Rank Matrix

- **Definition of Full Rank Matrix:**

- A matrix  $\mathbf{A}$  of dimensions  $m \times n$  is said to have **full rank** if the rank of  $\mathbf{A}$  is equal to the minimum of its number of rows  $m$  and columns  $n$ . Mathematically, this is expressed as:

$$\text{Rank}(\mathbf{A}) = \min(m, n)$$

- **Properties:**

- **Full Row Rank:** A matrix has full row rank if every row is linearly independent, which for  $m \leq n$ , means:

$$\text{Rank}(\mathbf{A}) = m$$

- **Full Column Rank:** A matrix has full column rank if every column is linearly independent, applicable when  $n \leq m$ , signified by:

$$\text{Rank}(\mathbf{A}) = n$$

- **Invertibility:** A square matrix  $m = n$  with full rank is invertible, which implies:

$$\text{Rank}(\mathbf{A}) = n = m$$

and  $\det(\mathbf{A}) \neq 0$ , indicating a non-zero determinant.

## Example 4: A 3x3 matrix that is not full rank

Consider the matrix:

$$D = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

- The rank of this matrix is 2 (since the rows are linearly dependent).
- Specifically, the third row can be written as a linear combination of the first two rows.

Thus, **matrix D is not full rank** because its rank (2) is less than the smaller dimension (which is 3).

# Pseudo Inverse Matrices

- Linear Algebra Concept:

- Definition of Pseudo Inverse Matrices:

- The pseudo inverse, or Moore-Penrose inverse, of a matrix  $\mathbf{A}$  is a matrix  $\mathbf{A}^+$  that generalizes the concept of an inverse to non-square matrices or matrices that are not full rank.

- Properties:

- $\mathbf{A}^+$  exists for any matrix  $\mathbf{A}$ , square or non-square.
- It satisfies the conditions:  $\mathbf{A}\mathbf{A}^+\mathbf{A} = \mathbf{A}$  and  $\mathbf{A}^+\mathbf{A}\mathbf{A}^+ = \mathbf{A}^+$ .

- Difference with Inverse Matrix:

- A regular inverse only exists for square, non-singular matrices.
- The pseudo inverse can be computed for any matrix, providing a solution (often the least squares solution) to  $\mathbf{A}\mathbf{x} = \mathbf{b}$  even when  $\mathbf{A}$  does not have a conventional inverse.

## Numerical Application:

- Example:

- Given  $\mathbf{A} = \begin{bmatrix} 2 & 4 \\ 1 & 3 \\ 0 & 0 \end{bmatrix}$  and  $\mathbf{b} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$ :

- Compute  $\mathbf{A}^+$  and use it to solve for  $\mathbf{x}$  in  $\mathbf{A}\mathbf{x} = \mathbf{b}$ , providing the least squares solution.

- Solution:

$$\mathbf{x} = \mathbf{A}^+\mathbf{b} = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$$

- This solution minimizes the squared error  $\|\mathbf{A}\mathbf{x} - \mathbf{b}\|$ .

- Calculating Pseudo Inverses:

- Use `np.linalg.pinv()` to compute the pseudo inverse of any matrix.

- Code:

```
python Copy code

import numpy as np
A = np.array([[2, 4], [1, 3], [0, 0]])
A_pinv = np.linalg.pinv(A)
```

- Using Pseudo Inverses to Solve Systems:

- Solve systems, particularly over-determined or under-determined systems.

- Code:

```
python Copy code

b = np.array([1, 2, 3])
x = np.dot(A_pinv, b)
print("Solution x:", x)
```

# Linear Regression for Sequence Labeling Using Pseudo-Inverses

- Context and Objective:

- Goal:** Apply linear regression to sequence labeling tasks, such as Named Entity Recognition (NER), to predict the presence of named entities in text snippets using a simple dataset.
- Challenge:** Managing over-determined systems where there are more observations (data points) than features (variables), which is common in text data analysis.

- Dataset and Preparation:

- Sample Dataset:

- A collection of text snippets and corresponding binary labels indicate whether each snippet contains a named entity (1 for presence, 0 for absence).

- Text Snippets:

- "John works at OpenAI."
- "Yesterday was very sunny."
- "She bought 300 shares of Tesla."
- "He loves to play soccer."

Labels:

•  $\mathbf{b} = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix}$  # 1 and 3 contain named entities.

python

Copy code

```
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer

# Text data and labels
snippets = ["John works at OpenAI.", "Yesterday was very sunny.", "She bought 300 shares of Tesla.", "He loves to play soccer."]
labels = np.array([1, 0, 1, 0])
```

# Linear Regression for Sequence Labeling Using Pseudo-Inverses

- **Feature Extraction:**

- Convert text snippets into a numerical format using TF-IDF, a method that transforms text into a feature vector that represents the importance of words in a document relative to a corpus.

- **Mathematical Formulation:**

- **Equation:**

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$$

- **A**: Feature matrix derived from text snippets.
- **x**: Weight vector to be determined.
- **b**: Known labels vector.

```
# Feature extraction
vectorizer = TfidfVectorizer(max_features=5) # Limit to 5 features
A = vectorizer.fit_transform(snippets).toarray()
```

```
Feature names: ['300' 'tesla' 'to' 'very' 'was']
Feature matrix (A):
[[0.          0.          0.          0.          0.          ]
 [0.          0.          0.          0.70710678  0.70710678]
 [0.70710678  0.70710678  0.          0.          0.          ]
 [0.          0.          1.          0.          0.          ]
 ... ..]
```

# Linear Regression for Sequence Labeling Using Pseudo-Inverses

- Pseudo-Inverse Computation:

- Since  $\mathbf{A}$  is typically over-determined (more rows than columns), compute the Moore-Penrose pseudo-inverse  $\mathbf{A}^+$  to find the least squares solution to the linear system:

$$\mathbf{A}^+ = (\mathbf{A}^T \mathbf{A})^{-1} \mathbf{A}^T$$

- Solve for  $\mathbf{x}$ :

$$\mathbf{x} = \mathbf{A}^+ \mathbf{b}$$

```
# Calculate the pseudo-inverse of A and solve for x
A_pinv = np.linalg.pinv(A)
x = np.dot(A_pinv, labels) # Solve for weights

print("Weight vector (x):", x)
```

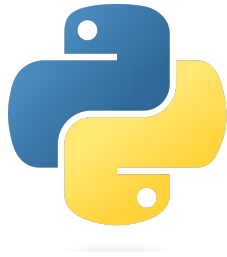
# Linear Regression for Sequence Labeling Using Pseudo-Inverses

```
1 import numpy as np
2 from sklearn.feature_extraction.text import TfidfVectorizer
3
4 # Define the text data and labels
5 snippets = [
6     "John works at OpenAI.",
7     "Yesterday was very sunny.",
8     "She bought 300 shares of Tesla.",
9     "He loves to play soccer."
10 ]
11 labels = np.array([1, 0, 1, 0])
12
13 # Initialize and fit the TF-IDF vectorizer
14 vectorizer = TfidfVectorizer(max_features=5) # Limit to 5 fe
15 A = vectorizer.fit_transform(snippets).toarray() # Feature m
16
17 # Display feature names and the feature matrix
18 print("Feature names:", vectorizer.get_feature_names_out())
19 print("Feature matrix (A):\n", A)
20
21 # Step 3: Computing the Pseudo-Inverse and Solving
22 # Compute the pseudo-inverse of A
23 A_pinv = np.linalg.pinv(A)
24 x = np.dot(A_pinv, labels) # Solve for x
25
26 print("Solution vector x:", x)
27
```

```
Feature names: ['300' 'tesla' 'to' 'very' 'was']
Feature matrix (A):
[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0.70710678 0.70710678]
 [0.70710678 0.70710678 0. 0. 0.]
 [0. 0. 1. 0. 0.]]
Solution vector x: [0.70710678 0.70710678 0. 0. 0.]
```



# Module 6: Systems of Linear Equations



# Module 6: Systems of Linear Equations

## 6.1 Solving Linear Systems



**CS316**  
**INTRODUCTION TO AI AND DATA  
SCIENCE**

**CHAPTER 3**  
**PANDAS AND NUMPY**

**LECTURE 1**  
*PANDAS*

# 5.1 Solving Linear Systems

- Representing Systems as  $\mathbf{Ax} = \mathbf{b}$ :

- A linear system can be represented in matrix form where  $\mathbf{A}$  is a matrix of coefficients,  $\mathbf{x}$  is a column vector of variables, and  $\mathbf{b}$  is the result vector.

- Compact Form:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

- Methods for Solving:

- Gaussian Elimination
- LU Decomposition
- Matrix Inversion (if  $\mathbf{A}$  is invertible)

## Gaussian Elimination with Numpy

- Solving Equations Using `np.linalg.solve`:

- This function is used to find vector  $\mathbf{x}$  such that  $\mathbf{Ax} = \mathbf{b}$ .
- Practical Example:

```
python

import numpy as np
A = np.array([[3, 1], [1, 2]])
b = np.array([9, 8])
x = np.linalg.solve(A, b)
print("Solution for Ax = b:", x)
```

- Output should show the values of  $\mathbf{x}$  that satisfy the equation.

# Solving Linear Systems with Gaussian Elimination

```
1  import numpy as np
2
3  # Define the matrix A and vector b
4  A = np.array([[3, 1], [1, 2]])
5  b = np.array([9, 8])
6
7  # Solve the linear system Ax = b
8  x = np.linalg.solve(A, b)
9
10 # Print the components and the solution
11 print("Matrix A:\n", A)
12 print("\nVector b:\n", b)
13 print("\nSolution vector x for Ax = b:\n", x)
14
```

Matrix A:

```
[[3 1]
 [1 2]]
```

Vector b:

```
[9 8]
```

Solution vector x for Ax = b:

```
[2. 3.]
```

# Gaussian Elimination

- **Definition:**

- A method for solving linear equations by transforming the system's matrix into an upper triangular form using row operations, from which the solutions can be derived through back-substitution.

- **Process:**

- Perform row operations to form a row-echelon matrix.
- Solve from the bottom row up (back-substitution).

- **NumPy Implementation:**

- NumPy does not have a direct function for Gaussian Elimination, but `numpy.linalg.solve` effectively uses this concept when applicable.

python

Copy code

```
import numpy as np
A = np.array([[2, 1, -1], [-3, -1, 2], [-2, 1, 2]])
b = np.array([8, -11, -3])
x = np.linalg.solve(A, b)
print("Solution:", x)
```

- **Math Equations:**

- Consider the system:

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = b_1$$

$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 = b_2$$

$$a_{31}x_1 + a_{32}x_2 + a_{33}x_3 = b_3$$

- **Gaussian Elimination Process:**

- Convert to REF:

$$\left[ \begin{array}{ccc|c} a'_{11} & a'_{12} & a'_{13} & b'_1 \\ 0 & a'_{22} & a'_{23} & b'_2 \\ 0 & 0 & a'_{33} & b'_3 \end{array} \right]$$

- Solve from the bottom up:

- $x_3 = \frac{b'_3}{a'_{33}}$

- $x_2 = \frac{b'_2 - a'_{23}x_3}{a'_{22}}$

- $x_1 = \frac{b'_1 - a'_{12}x_2 - a'_{13}x_3}{a'_{11}}$

# Gaussian Elimination: Numerical Example

- System of Linear Equations:

- Consider the system:

$$2x + 3y - z = 5$$

$$4x + y + 2z = 6$$

$$-2x + 5y - 3z = 8$$

- Step 1: Row Echelon Form (REF):

- Initial Matrix:

$$\left[ \begin{array}{ccc|c} 2 & 3 & -1 & 5 \\ 4 & 1 & 2 & 6 \\ -2 & 5 & -3 & 8 \end{array} \right]$$

# Gaussian Elimination: Numerical Example

- Step 1: Row Echelon Form (REF):

- Initial Matrix:

$$\left[ \begin{array}{ccc|c} 2 & 3 & -1 & 5 \\ 4 & 1 & 2 & 6 \\ -2 & 5 & -3 & 8 \end{array} \right]$$

- Operations:

- $R_2 = R_2 - 2 \times R_1$

$$\begin{aligned} (4 - 2 \times 2)x + (1 - 2 \times 3)y + (2 + 2 \times 1)z &= 6 - 2 \times 5 \\ 0x - 5y + 4z &= -4 \end{aligned}$$

- $R_3 = R_3 + R_1$

$$\begin{aligned} (-2 + 2)x + (5 + 3)y + (-3 - 1)z &= 8 + 5 \\ 0x + 8y - 4z &= 13 \end{aligned}$$

- Resulting Matrix:

$$\left[ \begin{array}{ccc|c} 2 & 3 & -1 & 5 \\ 0 & -5 & 4 & -4 \\ 0 & 8 & -4 & 13 \end{array} \right]$$

- Simplify further if necessary and attempt back substitution.

- Step 2: Back Substitution:

- Begin with the simplest equation (from the last row, if possible), and solve for each variable.
- However, as noted previously, this system leads to an inconsistency, illustrating that Gaussian elimination not only solves systems but also identifies no-solution scenarios.

- Solve from the bottom up:

- $x_3 = \frac{b'_3}{a'_{33}}$

- $x_2 = \frac{b'_2 - a'_{23}x_3}{a'_{22}}$

- $x_1 = \frac{b'_1 - a'_{12}x_2 - a'_{13}x_3}{a'_{11}}$

# LU Decomposition

- **Definition:**

- LU Decomposition splits a matrix **A** into two factors: a lower triangular matrix **L** and an upper triangular matrix **U**, such that **A** = **L** × **U**.

- **Mathematical Concept:**

- **Decomposition Equation:**

$$\mathbf{A} = \mathbf{L} \times \mathbf{U}$$

- **Where:**

- **L** is a lower triangular matrix with ones on the diagonal.
- **U** is an upper triangular matrix.

- **Expanded Form:**

- For a  $3 \times 3$  matrix **A**:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{bmatrix} \times \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

- **Application:**

- Efficiently solve systems of equations **Ax** = **b**, particularly when dealing with multiple right-hand sides or repeatedly using the same matrix **A** with different **b** vectors.

- Using `scipy.linalg.lu`:

- `scipy.linalg.lu` decomposes **A** into **L** and **U** and can be used to solve for **x** using forward and backward substitution.

```
python Copy code

import numpy as np
from scipy.linalg import lu

# Define the matrix A
A = np.array([[3, 2], [6, 4]])

# Perform LU decomposition
P, L, U = lu(A)
print("L (Lower Triangular Matrix):\n", L)
print("U (Upper Triangular Matrix):\n", U)
```

# LU Decomposition: Numerical Example

- Given Matrix **A**:

- Consider the matrix:

$$\mathbf{A} = \begin{bmatrix} 4 & 3 \\ 6 & 3 \end{bmatrix}$$

- Step-by-Step LU Decomposition:

1. Initialize:

- Assume the form  $\mathbf{A} = \mathbf{L} \times \mathbf{U}$  where  $\mathbf{L}$  is a lower triangular matrix with 1s on the diagonal, and  $\mathbf{U}$  is an upper triangular matrix.

- $\mathbf{L} = \begin{bmatrix} 1 & 0 \\ l_{21} & 1 \end{bmatrix}, \mathbf{U} = \begin{bmatrix} u_{11} & u_{12} \\ 0 & u_{22} \end{bmatrix}$

# LU Decomposition: Numerical Example

## 2. Decomposition:

- From  $\mathbf{A}_{11} = \mathbf{L}_{11}\mathbf{U}_{11}$ ,  $4 = 1 \times u_{11}$  so  $u_{11} = 4$ .
- From  $\mathbf{A}_{12} = \mathbf{L}_{11}\mathbf{U}_{12}$ ,  $3 = 1 \times u_{12}$  so  $u_{12} = 3$ .
- From  $\mathbf{A}_{21} = \mathbf{L}_{21}\mathbf{U}_{11}$ ,  $6 = l_{21} \times 4$  so  $l_{21} = 1.5$ .
- From  $\mathbf{A}_{22} = \mathbf{L}_{21}\mathbf{U}_{12} + \mathbf{L}_{22}\mathbf{U}_{22}$ ,  $3 = 1.5 \times 3 + 1 \times u_{22}$  leading to  $u_{22} = 0$ .

## • Resulting Matrices $\mathbf{L}$ and $\mathbf{U}$ :

- $\mathbf{L} = \begin{bmatrix} 1 & 0 \\ 1.5 & 1 \end{bmatrix}, \mathbf{U} = \begin{bmatrix} 4 & 3 \\ 0 & 0 \end{bmatrix}$

## • Verification:

- The product  $\mathbf{L} \times \mathbf{U}$  reconstructs  $\mathbf{A}$ , though in this example,  $\mathbf{U}$  has a zero row indicating a degenerate (singular) matrix, which typically signals a problem with the matrix being non-invertible or poorly conditioned for certain operations.

## • Given Matrix $\mathbf{A}$ :

- Consider the matrix:

$$\mathbf{A} = \begin{bmatrix} 4 & 3 \\ 6 & 3 \end{bmatrix}$$

## • Step-by-Step LU Decomposition:

### 1. Initialize:

- Assume the form  $\mathbf{A} = \mathbf{L} \times \mathbf{U}$  where  $\mathbf{L}$  is a lower triangular matrix with 1s on the diagonal, and  $\mathbf{U}$  is an upper triangular matrix.

- $\mathbf{L} = \begin{bmatrix} 1 & 0 \\ l_{21} & 1 \end{bmatrix}, \mathbf{U} = \begin{bmatrix} u_{11} & u_{12} \\ 0 & u_{22} \end{bmatrix}$

# LU Decomposition: Solving $Ax=b$

- **Initial Setup:**

- Given the matrix  $A$  and vector  $b$ , we first decompose  $A$  into  $L$  (lower triangular) and  $U$  (upper triangular):

$$A = LU$$

- For our previous example:

$$A = \begin{bmatrix} 4 & 3 \\ 6 & 3 \end{bmatrix}, \quad b = \begin{bmatrix} 24 \\ 30 \end{bmatrix}$$

- **Step 1: Decompose  $A$ :**

- Assume  $L = \begin{bmatrix} 1 & 0 \\ 1.5 & 1 \end{bmatrix}$  and  $U = \begin{bmatrix} 4 & 3 \\ 0 & 0 \end{bmatrix}$  from the decomposition step.

- **Step 2: Solve  $Ly = b$  (Forward substitution):**

- $Ly = b$  translates into:

$$\begin{bmatrix} 1 & 0 \\ 1.5 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} 24 \\ 30 \end{bmatrix}$$

- Solve for  $y_1$  and  $y_2$ :

- $y_1 = 24$
- $1.5 \times 24 + y_2 = 30 \Rightarrow y_2 = 30 - 36 = -6$

- **Step 3: Solve  $Ux = y$  (Back substitution):**

- $Ux = y$  translates into:

$$\begin{bmatrix} 4 & 3 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 24 \\ -6 \end{bmatrix}$$

- Normally, solve for  $x_2$  and  $x_1$ :

- This matrix shows an inconsistency or a special condition due to the zero row in  $U$ . This indicates that the system may be underdetermined or have infinitely many solutions depending on the consistency of the equations.

- **Conclusion:**

- In this case, the inconsistency due to the zero row in  $U$  means that the matrix  $A$  is singular, and the system does not have a unique solution.
- This example highlights a situation where the LU decomposition method exposes the properties of the matrix  $A$  that affect the solvability of  $Ax = b$ .

# Matrix Inversion

- **Definition:**

- Matrix Inversion involves calculating the inverse of a matrix  $\mathbf{A}$ , denoted as  $\mathbf{A}^{-1}$ , where  $\mathbf{A} \times \mathbf{A}^{-1} = \mathbf{I}$ , the identity matrix.

- **Mathematical Concept:**

- **Inversion Equation:**

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

- **Conditions:**

- $\mathbf{A}$  must be square (same number of rows and columns).
- $\mathbf{A}$  must be non-singular (determinant  $\neq 0$ ).

- **Math Expanded:**

- For a  $2 \times 2$  matrix  $\mathbf{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$ :

$$\mathbf{A}^{-1} = \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

- The determinant  $ad - bc$  should not be zero to ensure that the inverse exists.

- **Application:**

- Useful for solving linear systems  $\mathbf{Ax} = \mathbf{b}$  by transforming into  $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$ .
- Critical in algorithms that require matrix operations like finding solutions to linear equations, computer graphics transformations, and optimization problems.

- **NumPy Implementation:**

- Using `np.linalg.inv` to invert the matrix.

```
python Copy code

import numpy as np
A = np.array([[1, 2], [3, 4]])
b = np.array([5, 11])
A_inv = np.linalg.inv(A)
x = np.dot(A_inv, b)
print("Solution using Matrix Inversion:", x)
```

# Matrix Inversion: Solving $Ax=b$

Matrix Inversion: Numerical Example for Solving  $Ax = b$

- Given Matrix  $A$  and Vector  $b$ :

- Consider the matrix:

$$A = \begin{bmatrix} 3 & 4 \\ 2 & 1 \end{bmatrix}$$

- And the vector:

$$b = \begin{bmatrix} 24 \\ 10 \end{bmatrix}$$

- Objective:

- Solve for  $x$  in  $Ax = b$  using matrix inversion.

# Matrix Inversion: Solving $Ax=b$

- Step 1: Calculate the Inverse of  $A$ :
  - Formula for Inverse of a 2x2 Matrix:

$$\mathbf{A}^{-1} = \frac{1}{\det(\mathbf{A})} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

- Where  $a, b, c, d$  are the elements of  $A$ :

$$\det(\mathbf{A}) = (3 \times 1) - (4 \times 2) = 3 - 8 = -5$$

$$\mathbf{A}^{-1} = \frac{1}{-5} \begin{bmatrix} 1 & -4 \\ -2 & 3 \end{bmatrix} = \begin{bmatrix} -0.2 & 0.8 \\ 0.4 & -0.6 \end{bmatrix}$$

# Matrix Inversion: Solving $\mathbf{Ax}=\mathbf{b}$

- Step 2: Solve for  $\mathbf{x}$ :

- Using Matrix Inversion:

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b} = \begin{bmatrix} -0.2 & 0.8 \\ 0.4 & -0.6 \end{bmatrix} \begin{bmatrix} 24 \\ 10 \end{bmatrix}$$

- Calculating  $\mathbf{x}$ :

$$x_1 = (-0.2 \times 24) + (0.8 \times 10) = -4.8 + 8 = 3.2$$

$$x_2 = (0.4 \times 24) - (0.6 \times 10) = 9.6 - 6 = 3.6$$

$$\mathbf{x} = \begin{bmatrix} 3.2 \\ 3.6 \end{bmatrix}$$

# Matrix Inversion: Solving $Ax=b$

```
1 import numpy as np
2
3 # Define the matrix A
4 A = np.array([[3, 2],
5               [4, 1]])
6
7 # Define the vector b
8 b = np.array([24, 10])
9
10 # Check if matrix A is invertible by calculating its determinant
11 if np.linalg.det(A) != 0:
12     # Calculate the inverse of matrix A
13     A_inv = np.linalg.inv(A)
14
15     # Solve for x by multiplying the inverse of A with vector b
16     x = np.dot(A_inv.T, b)
17
18     # Print the components and the solution
19     print("Matrix A:\n", A)
20     print("\nInverse of Matrix A:\n", A_inv)
21     print("\nVector b:\n", b)
22     print("\nSolution vector x for Ax = b using A's inverse:\n", x)
23 else:
24     print("Matrix A is not invertible, cannot solve using the inverse.")
```

Matrix A:

```
[[3 2]
 [4 1]]
```

Inverse of Matrix A:

```
[[-0.2  0.4]
 [ 0.8 -0.6]]
```

Vector b:

```
[24 10]
```

Solution vector x for  $Ax = b$  using A's inverse:

```
[3.2 3.6]
```

# Solving a Linear System of Equations

- **Introduction:**
  - Understanding the relationship between matrix operations and their geometric interpretations can provide deeper insights into solving linear systems.
- **Matrix Formulation:**
  - **Matrix Representation:**
    - The system of equations can be represented in matrix form as:

$$\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 5 \\ 7 \end{bmatrix}$$

- Where  $\mathbf{A}$  is the coefficient matrix, and  $\mathbf{b}$  is the constant matrix.
- The equation system is:

$$\mathbf{Ax} = \mathbf{b}$$

# Solving a Linear System of Equations

- **Introduction:**
  - Understanding the relationship between matrix operations and their geometric interpretations can provide deeper insights into solving linear systems.
- **Matrix Formulation:**
  - **Matrix Representation:**
    - The system of equations can be represented in matrix form as:

$$\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 5 \\ 7 \end{bmatrix}$$

- Where  $\mathbf{A}$  is the coefficient matrix, and  $\mathbf{b}$  is the constant matrix.
- The equation system is:

$$\mathbf{Ax} = \mathbf{b}$$

# Solving a Linear System of Equations in 2D

- Translation to Linear Equations:

- Derived Equations:

- From the matrix form, the individual linear equations are:

- $2x_1 + x_2 = 5$  (From the first row of **A** and **b**)

- $x_1 + 3x_2 = 7$  (From the second row of **A** and **b**)

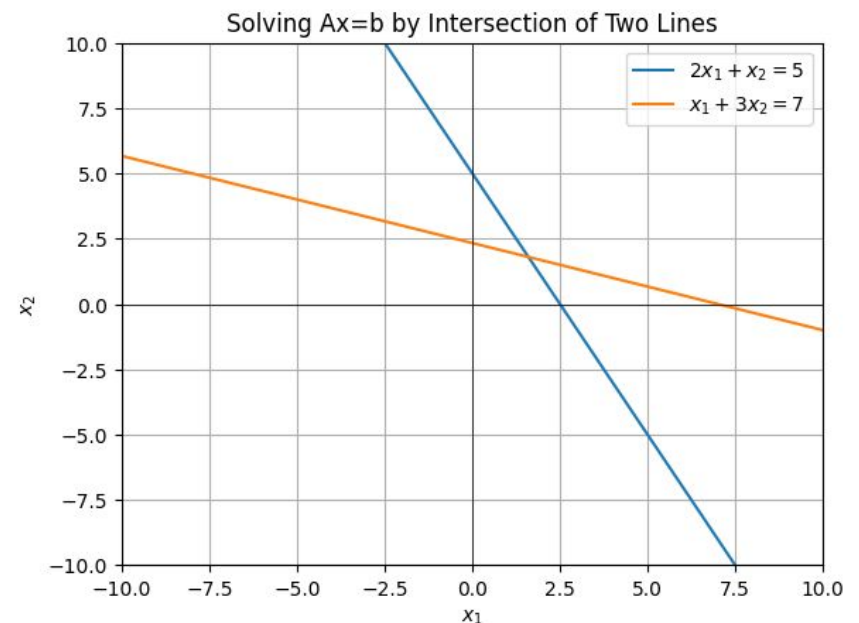
- Graphical Representation:

- Plotting Strategy:

- These equations can be graphed by rearranging each into  $y = mx + c$  form (where  $x_1$  is independent and  $x_2$  is dependent).

- Equation 1:  $x_2 = 5 - 2x_1$

- Equation 2:  $x_2 = \frac{7-x_1}{3}$



# Solving a Linear System of Equations in 3D

- **Introduction:**

- In a three-variable system, each linear equation can be represented as a plane in three-dimensional space. The solution to the system is the point or line where the planes intersect.

- **Equation Setup:**

- Consider a system of three equations:

- $x + 2y + z = 4$
- $y - z = 0$
- $2x + y + z = 5$

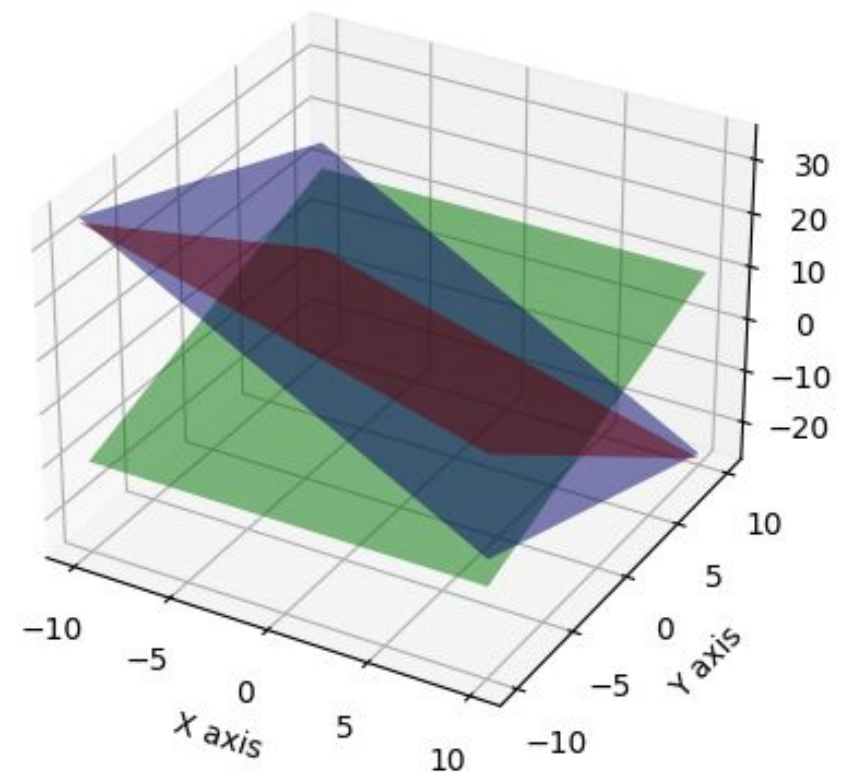
- **Graphical Representation in 3D:**

- Each equation can be graphed as a plane in a 3D space defined by axes  $x$ ,  $y$ , and  $z$ .

- **Plane Equations Derived:**

- **Plane 1:**  $z = 4 - x - 2y$
- **Plane 2:**  $z = y$
- **Plane 3:**  $z = 5 - 2x - y$

Solving  $Ax=b$  in 3D by Intersecting Planes



# Case Study: Linear Regression

- Context: Predicting House Prices
  - Problem Statement:
    - Use linear regression to predict house prices based on features such as area, number of bedrooms, and age of the house.

- Equation Setup:

- Represent the problem as  $\mathbf{Ax} = \mathbf{b}$ , where:

$\mathbf{A}$  = Feature Matrix,  $\mathbf{x}$  = Weight Vector,  $\mathbf{b}$  = Price Vector

- Feature Matrix  $\mathbf{A}$ :

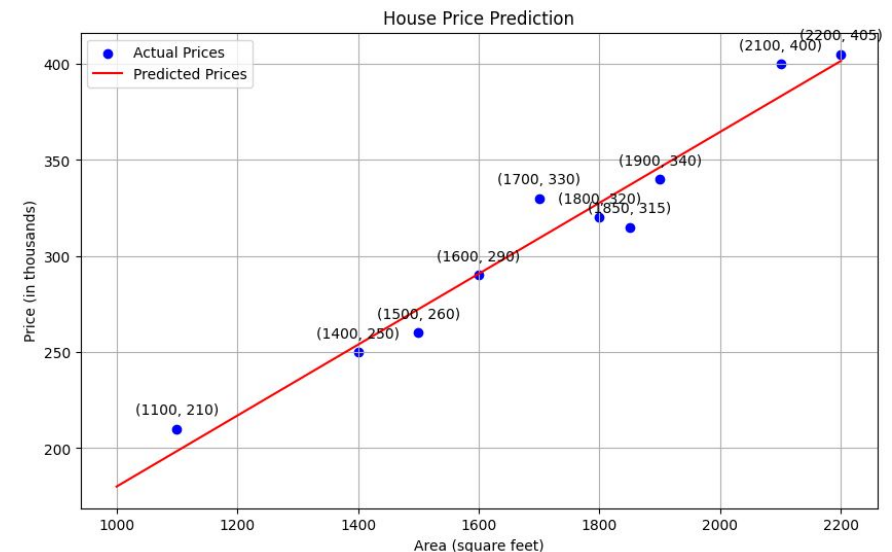
- Rows correspond to houses, columns to features:

$$\mathbf{A} = \begin{bmatrix} 1 & \text{Area} & \text{Bedrooms} & \text{Age} \\ 1 & 2100 & 5 & 10 \\ 1 & 1400 & 3 & 3 \\ 1 & 1800 & 4 & 8 \end{bmatrix}$$

- Price Vector  $\mathbf{b}$ :

- Prices of the houses in thousands:

$$\mathbf{b} = \begin{bmatrix} 400 \\ 250 \\ 320 \end{bmatrix}$$



# Case Study: Linear Regression

- **Context: Predicting House Prices**

- **Problem Statement:**

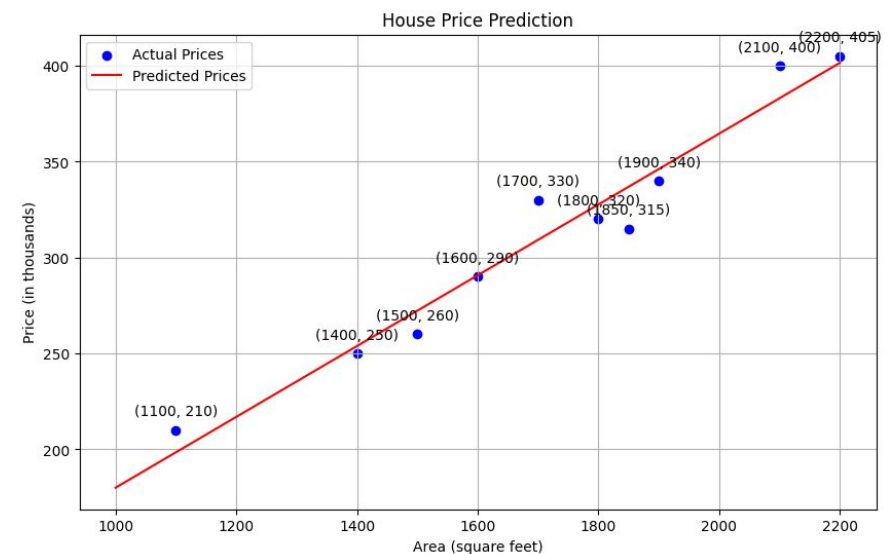
- Use linear regression to predict house prices based on features such as area, number of bedrooms, and age of the house.

- **Objective:**

- Solve for  $\mathbf{x}$ , the weights that best relate the features to the prices, using least squares to minimize the difference  $\|\mathbf{Ax} - \mathbf{b}\|$ .
- This illustrates how linear algebra powers fundamental data science tasks, specifically through techniques like matrix operations and linear systems.

```
1 import numpy as np
2 A = np.array([
3     [1, 2100], [1, 1400], [1, 1800], [1, 1500], [1, 1900],
4     [1, 1700], [1, 1600], [1, 1850], [1, 2200], [1, 1100]
5 ])
6 b = np.array([400, 250, 320, 260, 340, 330, 290, 315, 405, 210])
7 x, residuals, rank, s = np.linalg.lstsq(A, b, rcond=None)
8 print("Weight vector (x):", x)
9
```

Weight vector (x): [-4.58173935 0.18459577]



# Application of Linear Algebra in NLP: Spam Detection

- **Context and Objective:**

- **Goal:** Use linear algebra to classify emails as "spam" or "not spam" based on text features.
- **Application:** Text classification in Natural Language Processing (NLP).

- **Mathematical Formulation:**

- **Feature Matrix ( $\mathbf{A}$ ):**
  - Rows represent emails.
  - Columns represent features (e.g., frequency of keywords like "free", "winner").
- **Weight Vector ( $\mathbf{x}$ ):**
  - Contains weights for each feature determining their influence on the classification.
- **Target Vector ( $\mathbf{b}$ ):**
  - Binary labels indicating spam (1) or not spam (0).

- **Equation:**

$$\mathbf{Ax} = \mathbf{b}$$

- Solve for  $\mathbf{x}$  using least squares to minimize prediction errors:

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

- **Practical Example:**

- An email with higher frequencies of "free" might be classified as spam based on the model's learned weights.

- **NumPy Implementation (Simplified):**

```
python Copy code

import numpy as np
# Define a small feature matrix and target vector
A = np.array([[0.05, 0.9], [0.02, 0.1], [0.1, 0.8], [0.01, 0.05]])
b = np.array([1, 0, 1, 0])
# Compute weights
x = np.linalg.lstsq(A, b, rcond=None)[0]
```

# Application of Linear Algebra in NLP: Spam Detection

```
1 import numpy as np
2 from sklearn.feature_extraction.text import TfidfVectorizer
3 from sklearn.preprocessing import LabelEncoder
4
5 # Sample data (emails)
6 emails = [
7     "Free money now!!! Click here.",
8     "Hi Bob, how about a game of golf tomorrow?",
9     "Act now, and earn millions without leaving home.",
10    "Please call your mother.",
11    "Exclusive offer: earn rewards rapidly with our new credit cards."
12 ]
13
14 # Labels for the emails (1 for spam, 0 for not spam)
15 labels = [1, 0, 1, 0, 1]
16
17 # Step 1: Feature extraction using TF-IDF
18 vectorizer = TfidfVectorizer(stop_words='english', max_features=10)
19 A = vectorizer.fit_transform(emails).toarray()
20
21 # Step 2: Encode labels
22 le = LabelEncoder()
23 b = le.fit_transform(labels)
24
25 # Step 3: Solving for weights x in Ax = b using least squares
26 # Assume X is matrix A, and y is vector b
27 x, residuals, rank, s = np.linalg.lstsq(A, b, rcond=None)
28
29 # Output results
30 print("Features names:", vectorizer.get_feature_names_out())
31 print("Weight vector (x):", x)
32 print("Residuals:", residuals)
```

```
Features names: ['act' 'earn' 'home' 'leaving' 'millions' 'money' 'mother' 'new' 'offer'
'rapidly']
Weight vector (x): [0.40043879 0.68771373 0.40043879 0.40043879 0.40043879 1.
0.
0.45196465 0.45196465 0.45196465]
Residuals: []
```

```
1 # New emails to classify
2 new_emails = [
3     "Get your free trial now!!!",
4     "Meeting rescheduled to next week.",
5     "Make money fast by investing in stocks!"
6 ]
7
8 # Transform new emails into feature space using the fitted vectorizer
9 new_features = vectorizer.transform(new_emails).toarray()
10
11 # Predict using the linear model (dot product of features and weights)
12 predictions = np.dot(new_features, x)
13
14 # Apply a threshold to classify emails as spam or not
15 spam_threshold = 0.5 # Adjust based on model tuning and validation
16 predicted_labels = ['spam' if score > spam_threshold else 'not spam' for score in predictions]
17
18 # Output predictions
19 for email, label in zip(new_emails, predicted_labels):
20     print(f"Email: {email}\nClassified as: {label}\n")
21
22 # Output model details
23 print("Features names:", vectorizer.get_feature_names_out())
24 print("Weight vector (x):", x)
25 print("Residuals:", residuals)
```

```
Email: Get your free trial now!!!
Classified as: not spam
```

```
Email: Meeting rescheduled to next week.
Classified as: not spam
```

```
Email: Make money fast by investing in stocks!
Classified as: spam
```



# CS316 INTRODUCTION TO AI AND DATA SCIENCE

## CHAPTER 4 Integrating Linear Algebra with NumPy: Practical Foundations for Data Science

### LECTURE 3 *Eigenvalues and Eigenvectors* *PCA Case Study*

Prof. Anis Koubaa

October 2024

# Introduction to Matrix-Vector Multiplication

- **Title:** Understanding Matrix-Vector Multiplication
- **Content:**
  - Brief introduction to matrices and vectors.
  - Definition of matrix-vector multiplication.
  - Basic mathematical representation:  $\mathbf{A}v = w$ 
    - Where  $\mathbf{A}$  is a matrix,  $v$  is a vector, and  $w$  is the transformed vector.

# Transforming Vector Orientation

## 1. Explanation:

- Multiplying by a matrix can rotate a vector in the plane.
- The rotation matrix  $R$  rotates a vector by an angle  $\theta$  in a counterclockwise direction.

## 2. Example with a 2x2 Rotation Matrix:

- Matrix  $R$  for rotation by  $\theta$  degrees:

$$R = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

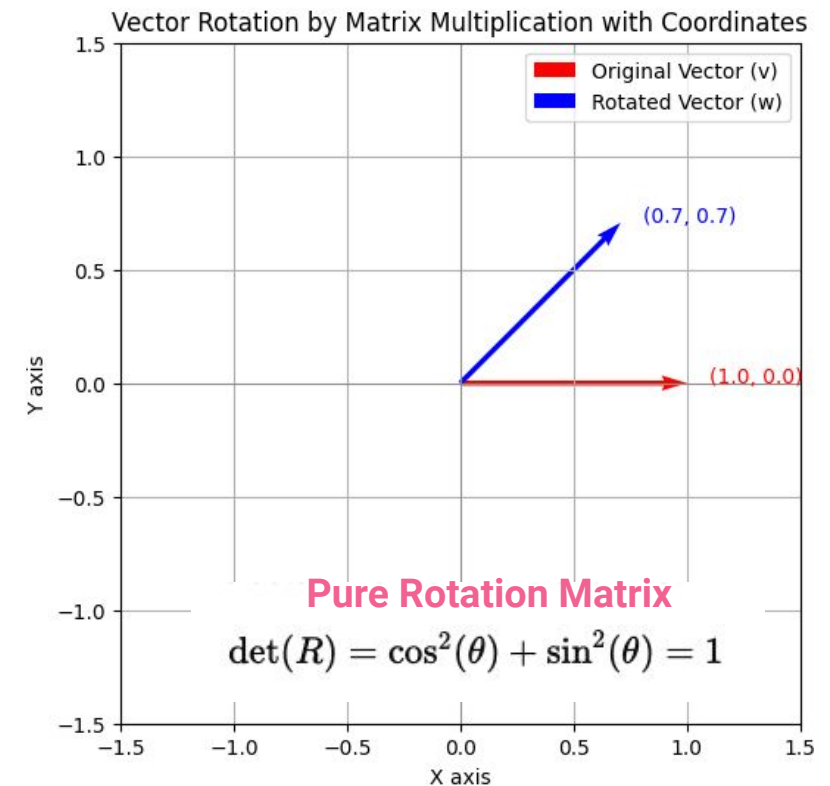
- Vector  $v$ :

$$v = \begin{bmatrix} x \\ y \end{bmatrix}$$

## 3. Numerical Example:

- Given:  $\theta = 45^\circ$ ,  $v = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$
- Calculation:

$$w = Rv = \begin{bmatrix} \cos(45^\circ) & -\sin(45^\circ) \\ \sin(45^\circ) & \cos(45^\circ) \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} \end{bmatrix}$$



# Scaling Vectors with Matrices

## 1. Explanation:

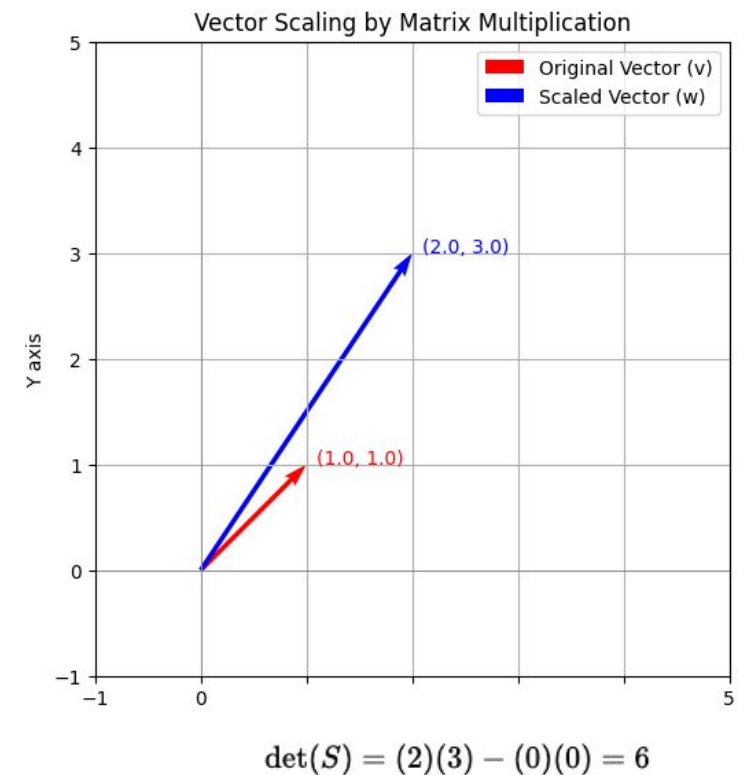
- Discuss how multiplying a vector by a scaling matrix changes its magnitude without altering its direction.
- Scaling matrices multiply each component of the vector by a scaling factor, allowing for differential scaling along different axes.

## 2. Introduction to Scaling Matrices:

- Scaling Matrix  $S$  for different scaling factors  $s_x$  and  $s_y$ :

$$S = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$$

- Where  $s_x$  and  $s_y$  are the scaling factors for the x and y components of the vector, respectively.



# Scaling Vectors with Matrices

## 3. Numerical Example:

- Given:

- Vector  $v = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$
- Scaling factors  $s_x = 2$  and  $s_y = 3$

- Calculation:

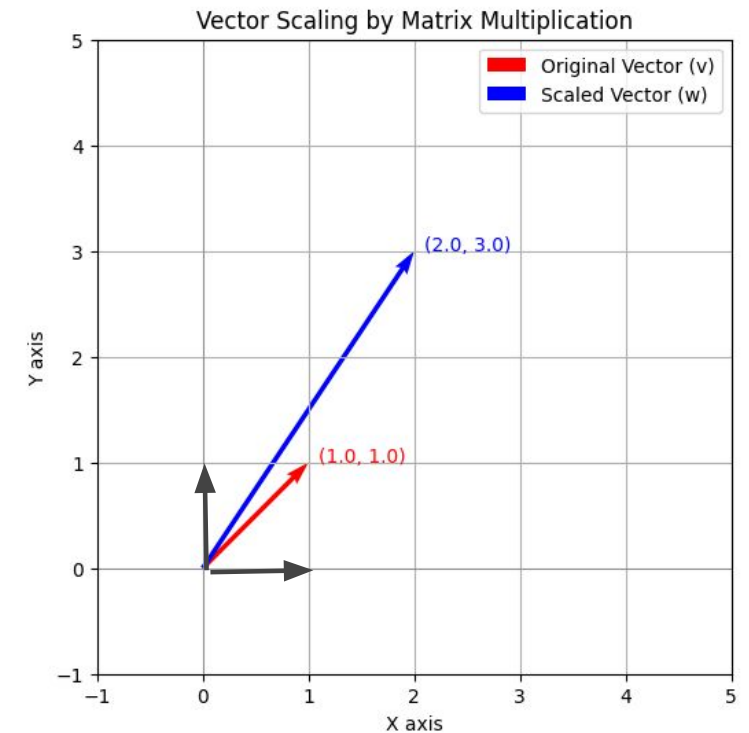
- Applying  $S$  to  $v$ :

$$w = Sv = \begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \end{bmatrix}$$

- The vector  $v$  is scaled to  $w$ , changing its length but not its orientation relative to the axes.



$$\det(S) = (2)(3) - (0)(0) = 6$$



# Scaling Vectors with a Scalar

## 1. Explanation:

- Describe how multiplying a vector by a scalar changes the vector's length but keeps its orientation (direction) unchanged.
- Scalars stretch or compress the magnitude of a vector without altering the angle it makes with the axes.

## 2. Mathematical Formulation:

### • Given:

- Vector  $v = \begin{bmatrix} x \\ y \end{bmatrix}$

- Scalar  $\alpha$

### • Scalar Multiplication:

- $w = \alpha v = \alpha \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \alpha x \\ \alpha y \end{bmatrix}$

### 3. Numerical Example:

#### • Given:

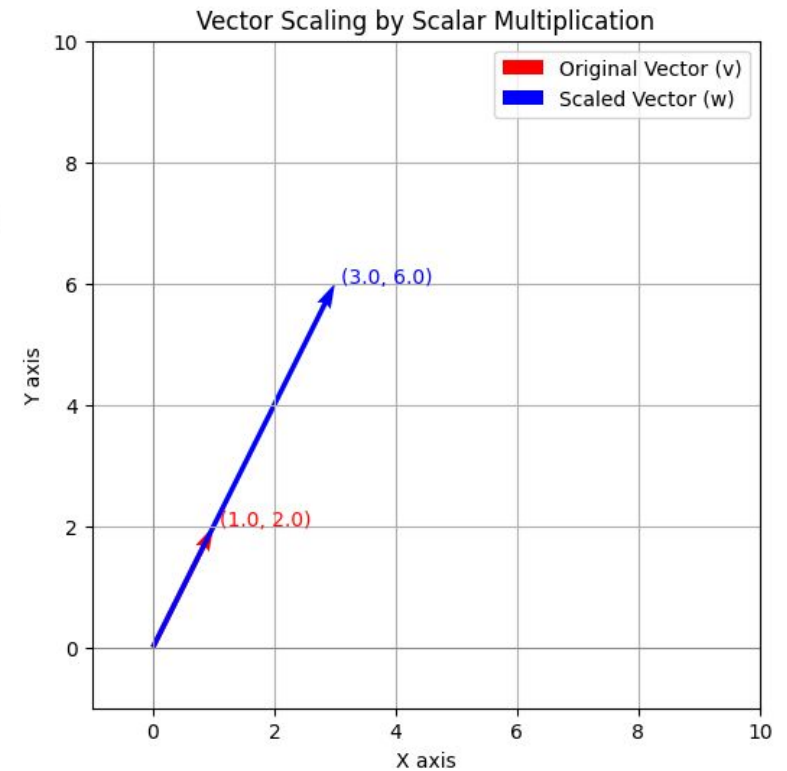
- Vector  $v = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$

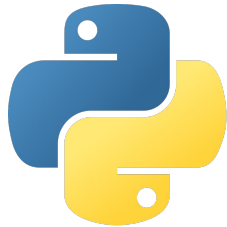
- Scalar  $\alpha = 3$

#### • Result:

- $w = 3 \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 3 \\ 6 \end{bmatrix}$

- The scaled vector  $w$  is three times longer than  $v$  but points in the same direction.





# Module 7: EigenValues and EigenVectors

## 7.1 Intuition



SPECIALIZATION

**CS316**  
**INTRODUCTION TO AI AND DATA  
SCIENCE**

**CHAPTER 3**  
**PANDAS AND NUMPY**

**LECTURE 1**

*PANDAS*

# Scaling Vectors with Matrices

## 3. Numerical Example:

- Given:

- Vector  $v = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$
- Scaling factors  $s_x = 2$  and  $s_y = 3$

- Calculation:

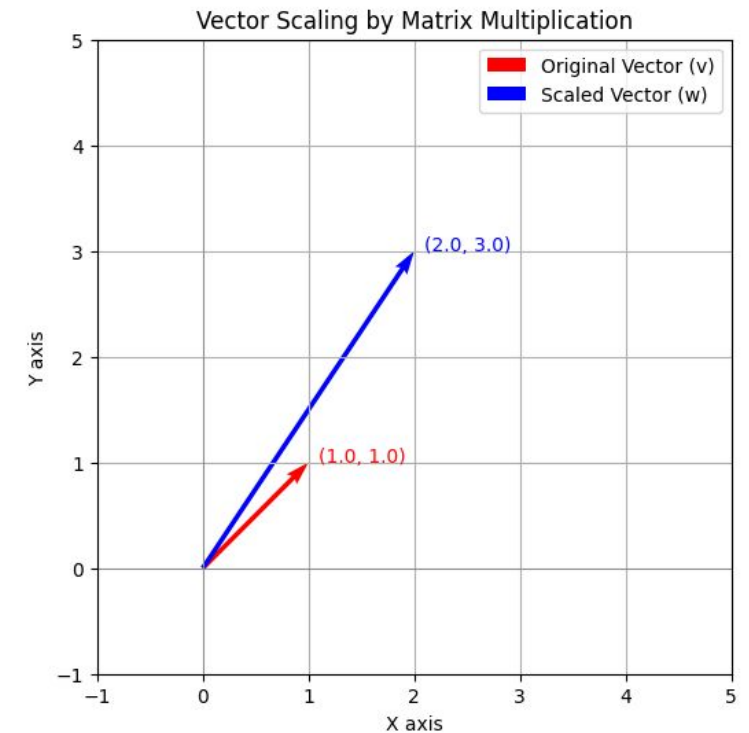
- Applying  $S$  to  $v$ :

$$w = Sv = \begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \end{bmatrix}$$

- The vector  $v$  is scaled to  $w$ , changing its length but not its orientation relative to the axes.



$$\det(S) = (2)(3) - (0)(0) = 6$$



# Scaling Vectors with Matrices

- Given:

- Vector  $v = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$
- Scaling factors  $s_x = 2$  and  $s_y = 3$

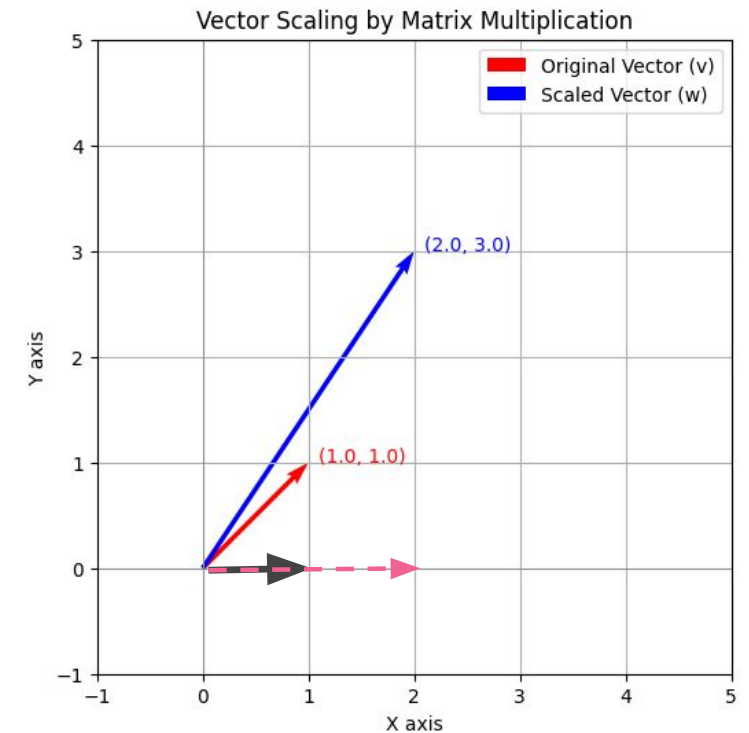
- Calculation:

- Applying  $S$  to  $v$ :

$$w = Sv = \begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \end{bmatrix}$$

- Interpretation:

- The vector  $v$  is scaled to  $w$ , changing its length but maintaining its direction along the x-axis.



# Scaling Vectors with Matrices

- Given:

- Vector  $v = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$
- Scaling factors  $s_x = 2$  and  $s_y = 3$

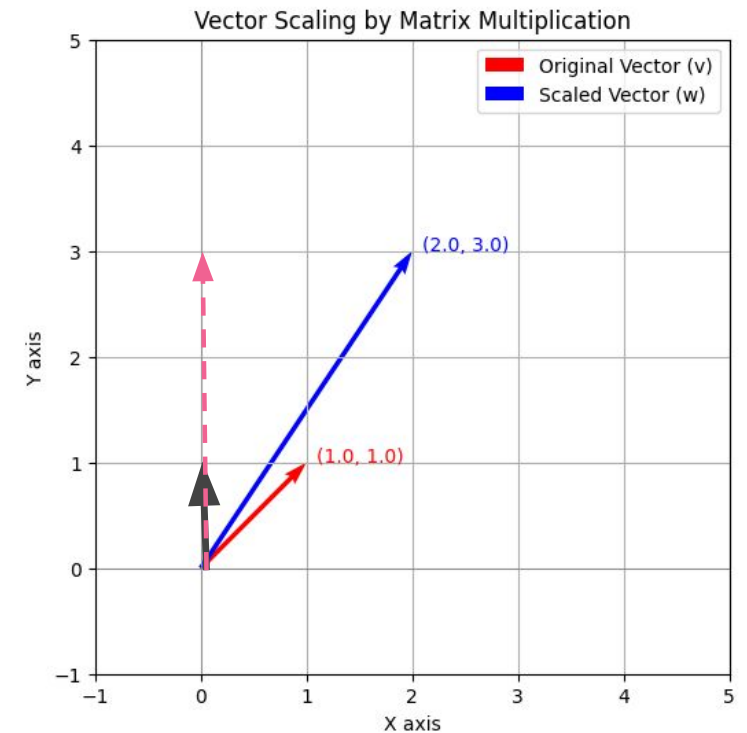
- Calculation:

- Applying  $S$  to  $v$ :

$$w = Sv = \begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 3 \end{bmatrix}$$

- Interpretation:

- The vector  $v$  is scaled to  $w$ , changing its length along the y-axis but maintaining its direction along the y-axis.



# Eigenvectors and Eigenvalues in Matrix Scaling

## Eigenvectors:

- Vectors that remain in the same direction after a matrix transformation.
- In the scaling matrix:

$$S = \begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix}$$

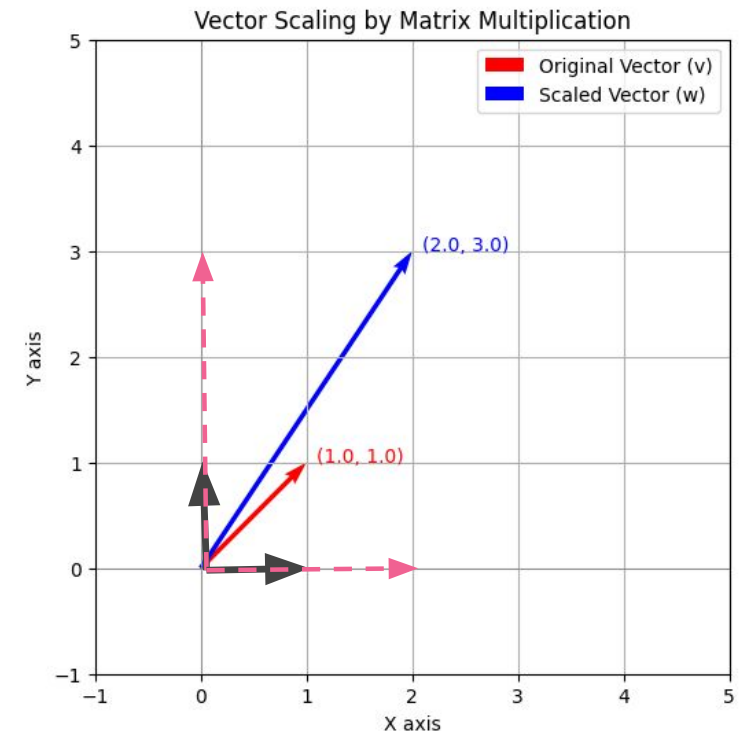
The vectors  $[1, 0]$  and  $[0, 1]$  are eigenvectors.

## Eigenvalues:

- Scalars by which the eigenvectors are scaled.
- For the scaling matrix  $S$ :
  - Eigenvector  $v_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$  has eigenvalue  $\lambda_1 = 2$ .
  - Eigenvector  $v_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$  has eigenvalue  $\lambda_2 = 3$ .

## Conclusion:

- The matrix  $S$  scales the eigenvector  $[1, 0]$  by 2 and  $[0, 1]$  by 3.
- These eigenvectors retain their original direction, and the eigenvalues represent the factors by which they are stretched.



# Impact of the Largest Eigenvalue in Transformations

## Key Concept:

- **Dominant Eigenvalue:** The eigenvalue with the highest magnitude has the greatest effect on scaling the corresponding eigenvector.
- In the matrix:

$$S = \begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix}$$

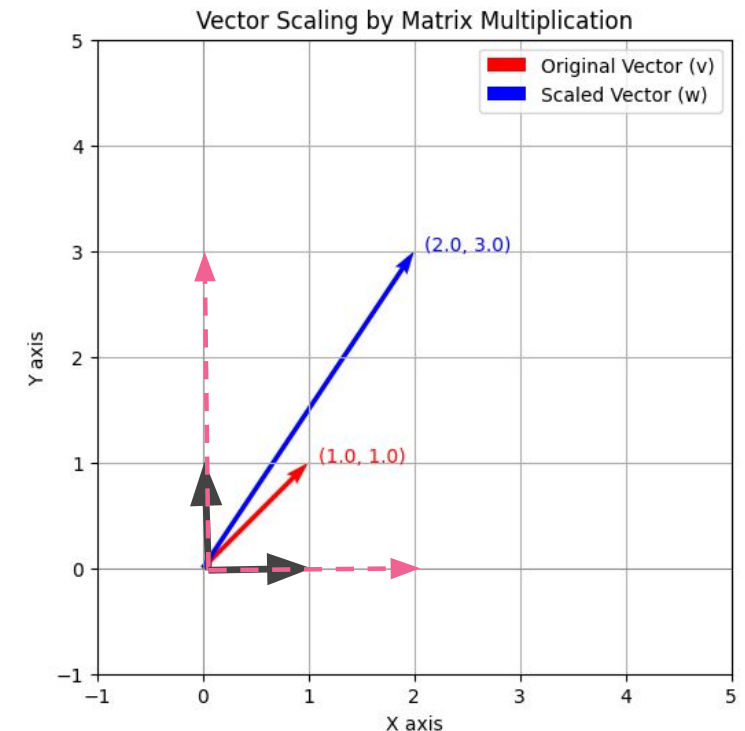
- Eigenvalue  $\lambda_2 = 3$  (corresponding to  $v_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ ) is the largest.
- This means the scaling along the y-axis has a **greater impact** than scaling along the x-axis.

## Application in Data Science:

- In **Principal Component Analysis (PCA)**:
  - The direction associated with the largest eigenvalue (principal component) captures the most variance in the data.
  - The principal components with larger eigenvalues have a **greater influence** on the dataset's structure, reducing dimensionality while retaining the most significant information.

## Conclusion:

- The largest eigenvalue not only represents the greatest scaling effect but also plays a key role in identifying key patterns and structures in data.



# Eigenvalues and Eigenvectors Explained Simply

## Imagine a Transformation:

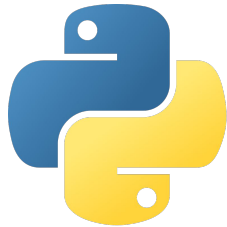
- Think of a transformation like stretching, squeezing, or rotating objects in space.
- In math, we represent these transformations using **matrices** that act on **vectors**.

## Special Vectors That Keep Their Direction:

- When we apply a transformation to most vectors, they change direction and length.
- However, **eigenvectors** are special because they **do not change direction** when transformed.
- They may get stretched (longer), compressed (shorter), or flipped (direction reversed), but they still point along the same line.

## Scaling Factors—Eigenvalues:

- The amount by which an eigenvector is stretched or compressed is called an **eigenvalue**.
- If the eigenvalue is:
  - **Greater than 1**: The eigenvector is stretched.
  - **Between 0 and 1**: The eigenvector is compressed.
  - **Negative**: The eigenvector flips direction.
  - **Zero**: The eigenvector collapses to a point (uncommon in practical scenarios).



## Module 7: EigenValues and EigenVectors

### 7.1 Formal Definitions



SPECIALIZATION

**CS316**  
**INTRODUCTION TO AI AND DATA  
SCIENCE**

**CHAPTER 3**  
**PANDAS AND NUMPY**

**LECTURE 1**

*PANDAS*

# Introduction to Eigenvalues and Eigenvectors

## Eigenvalues and Eigenvectors

- **Eigenvalues ( $\lambda$ ):** Scale factors that alter the magnitude of eigenvectors during a transformation.

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

- **Eigenvectors ( $\vec{v}$ ):** Vectors that, when multiplied by a matrix  $A$ , change only in scale, not direction.

## Importance in Linear Algebra

- **Core Concept:**
  - Fundamental to understanding matrix behavior.
  - Indicative of system properties like stability and oscillation.
- **Applications:**
  - Essential in disciplines from physics to economics, crucial for algorithms in AI like dimensionality reduction.

- **Matrix A:**

$$\begin{pmatrix} 4 & 1 & 2 \\ 1 & 3 & 1 \\ 2 & 1 & 3 \end{pmatrix}$$

- **Eigenvalues:**

- Eigenvalue 1: 6.1819
- Eigenvalue 2: 1.4116
- Eigenvalue 3: 2.4064

- **Eigenvectors:**

- Eigenvector 1:  $[-0.7118, -0.4042, -0.5744]$
- Eigenvector 2:  $[-0.5665, -0.1531, 0.8097]$
- Eigenvector 3:  $[0.4153, -0.9018, 0.1200]$

- **Verification of  $A \cdot v = \lambda \cdot v$  for the first eigenvalue and eigenvector:**

- $A \cdot v = [-4.4002, -2.4989, -3.5511]$
- $\lambda \cdot v = [-4.4002, -2.4989, -3.5511]$

# NumPy for Eigenvalues and Eigenvectors

python

 Copy code

```
import numpy as np

# Define a square matrix
A = np.array([[4, 1], [1, 4]])

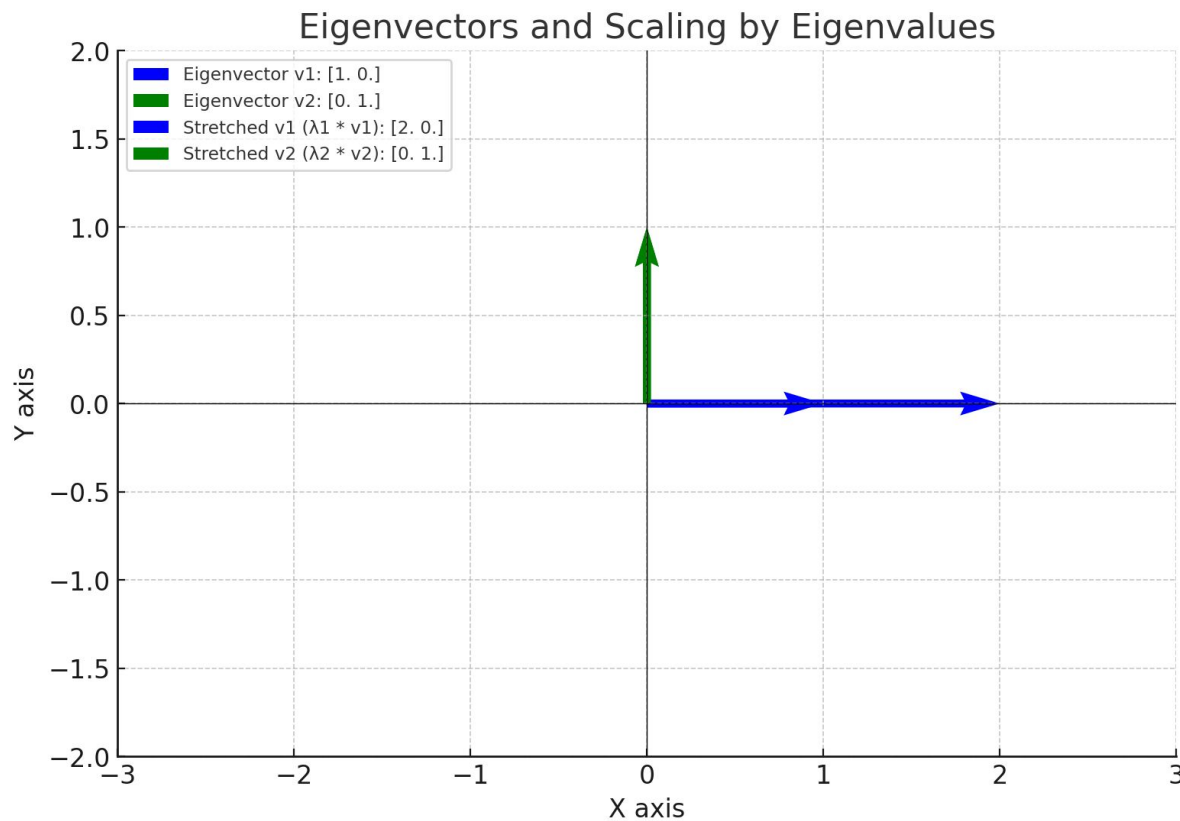
# Compute eigenvalues and eigenvectors
eigenvalues, eigenvectors = np.linalg.eig(A)

print("Eigenvalues:", eigenvalues)
print("Eigenvectors:\n", eigenvectors)
```

## Explanation

- **Matrix  $A$ :** Represents the system or transformation we are analyzing.
- **Function `np.linalg.eig`:** Calculates the eigenvalues and eigenvectors of matrix  $A$ .
  - **Eigenvalues:** Indicate the magnitude of stretching.
  - **Eigenvectors:** Show the directions that remain unchanged (except for scaling) under the transformation.

# NumPy for Eigenvalues and Eigenvectors



- The **blue and green** vectors represent the original eigenvectors.
- The **scaled (stretched)** vectors show how the eigenvectors are scaled by their respective eigenvalues.

Additionally, the printed values are as follows:

- **Eigenvalue 1: 2.00, Eigenvector 1: [1.0.]**
- **Eigenvalue 2: 1.00, Eigenvector 2: [0.1.]**

# Intuition Behind Eigenvalues and Eigenvectors

```
2 import numpy as np
3
4 # Define a 3x3 matrix
5 A = np.array([[4, 1, 2],
6              [1, 3, 1],
7              [2, 1, 3]])
8
9 # Compute the eigenvalues and eigenvectors of the matrix A
10 eigenvalues, eigenvectors = np.linalg.eig(A)
11
12 # Print eigenvalues and eigenvectors
13 print("Matrix A:")
14 print(A)
15 print("\nEigenvalues:")
16 for i, eigenvalue in enumerate(eigenvalues):
17     print(f"Eigenvalue {i+1}: {eigenvalue}")
18
19 print("\nEigenvectors:")
20 for i, eigenvector in enumerate(eigenvectors.T): # Transpose
21     print(f"Eigenvector {i+1}: {eigenvector}")
22
23 # Verifying  $A * v = \lambda * v$  for the first eigenvalue and eigenvector
24 v = eigenvectors[:, 0] # First eigenvector
25 lambda_ = eigenvalues[0] # First eigenvalue
26
27 # Compute  $A * v$ 
28 A_v = A @ v
29
30 # Compute  $\lambda * v$ 
31 lambda_v = lambda_ * v
32
33 # Print results
34 print("\nVerification of  $A * v = \lambda * v$  for the first eigenvalue and eigenvector:")
35 print("A * v =")
36 print(A_v)
37 print("\n $\lambda * v =$ ")
```

Matrix A:

```
[[4 1 2]
 [1 3 1]
 [2 1 3]]
```

Eigenvalues:

Eigenvalue 1: 6.181943336052388

Eigenvalue 2: 1.4116360093148959

Eigenvalue 3: 2.406420654632711

Eigenvectors:

Eigenvector 1: [-0.71178541 -0.40422217 -0.57442663]

Eigenvector 2: [-0.5664975 -0.15312282 0.80971228]

Eigenvector 3: [ 0.41526149 -0.90175265 0.12000026]

Verification of  $A * v = \lambda * v$  for the first eigenvalue and eigenvector:

$A * v =$

```
[-4.4002171 -2.49887857 -3.55107291]
```

$\lambda * v =$

```
[-4.4002171 -2.49887857 -3.55107291]
```

# Mathematical Representation

## Mathematical Expression

- Equation:

For matrix  $A$  and vector  $\vec{v}$  :

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

- Expanded Form:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \lambda \begin{bmatrix} x \\ y \end{bmatrix}$$

- Numerical Application:

- Example with specific values:

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

Eigenvalues: 3, 2

# The Trace of a Matrix and its Connection to Eigenvalues

## Definition: Trace of a Matrix:

- The **trace** of an  $n \times n$  matrix  $A$  is the sum of the elements on its diagonal.

$$\text{Tr}(A) = \sum_{i=1}^n A_{ii}$$

- Example: For matrix  $A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$ , the trace is:

$$\text{Tr}(A) = a_{11} + a_{22}$$

## Relationship to Eigenvalues:

- For an  $n \times n$  matrix  $A$ , the trace is also the sum of its eigenvalues  $\lambda_1, \lambda_2, \dots, \lambda_n$ :

$$\text{Tr}(A) = \lambda_1 + \lambda_2 + \dots + \lambda_n$$

- Example:

- Matrix  $S = \begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix}$
- Eigenvalues:  $\lambda_1 = 2, \lambda_2 = 3$
- Trace:  $\text{Tr}(S) = 2 + 3 = 5$

## Conclusion:

- The trace provides a simple way to understand the sum of the eigenvalues of a square matrix, and it plays a key role in many applications of linear algebra and data science.

# Trace and Eigenvalues of a Non-Diagonal 3x3 Matrix

## Numerical Example:

- Given the matrix  $B$ :

$$B = \begin{bmatrix} 4 & 2 & 1 \\ 2 & 3 & 1 \\ 1 & 1 & 2 \end{bmatrix}$$

## Trace of the Matrix:

- The trace is the sum of the diagonal elements:

$$\text{Tr}(B) = 4 + 3 + 2 = 9$$

## Eigenvalues:

- The eigenvalues of this matrix are:

- $\lambda_1 \approx 6.05$
- $\lambda_2 \approx 1.64$
- $\lambda_3 \approx 1.31$

## Conclusion:

- The trace is equal to the sum of the eigenvalues:

$$\text{Tr}(B) = \lambda_1 + \lambda_2 + \lambda_3 = 6.05 + 1.64 + 1.31 = 9$$

```
1 # Define a new 3x3 matrix with real eigenvalues
2 B = np.array([[4, 2, 1],
3               [2, 3, 1],
4               [1, 1, 2]])
5
6 # Calculate the trace of the new matrix
7 trace_B = np.trace(B)
8
9 # Calculate the eigenvalues of the new matrix
10 eigenvalues_B, _ = eig(B)
11
12 # Display results with better print formatting
13 print("Matrix B:")
14 print(B)
15 print("\nTrace of B (sum of diagonal elements):")
16 print(f"Trace(B) = {trace_B}")
17
18 print("\nEigenvalues of B:")
19 for i, eigenvalue in enumerate(eigenvalues_B):
20     print(f"Eigenvalue {i+1}: {eigenvalue:.4f}")
21
22 # Calculate the sum of eigenvalues
23 sum_of_eigenvalues_B = np.sum(eigenvalues_B)
24
25 # Print the sum of eigenvalues and compare with trace
26 print(f"\nSum of Eigenvalues: {sum_of_eigenvalues_B:.4f}")
27 print(f"Trace of B: {trace_B}")
28
29 # Check if they are approximately equal
30 if np.isclose(sum_of_eigenvalues_B, trace_B):
31     print("\nThe sum of the eigenvalues is approximately equal to the trace of the matrix.")
32 else:
33     print("\nThe sum of the eigenvalues is not equal to the trace of the matrix.")
```

# Trace and Eigenvalues of a Non-Diagonal 3x3 Matrix

Matrix B:  
[[4 2 1]  
 [2 3 1]  
 [1 1 2]]

Trace of B (sum of diagonal elements):  
Trace(B) = 9

Eigenvalues of B:  
Eigenvalue 1: 6.0489  
Eigenvalue 2: 1.6431  
Eigenvalue 3: 1.3080

Sum of Eigenvalues: 9.0000  
Trace of B: 9

The sum of the eigenvalues is approximately equal to the trace of the matrix.

```
1 # Define a new 3x3 matrix with real eigenvalues
2 B = np.array([[4, 2, 1],
3               [2, 3, 1],
4               [1, 1, 2]])
5
6 # Calculate the trace of the new matrix
7 trace_B = np.trace(B)
8
9 # Calculate the eigenvalues of the new matrix
10 eigenvalues_B, _ = eig(B)
11
12 # Display results with better print formatting
13 print("Matrix B:")
14 print(B)
15 print("\nTrace of B (sum of diagonal elements):")
16 print(f"Trace(B) = {trace_B}")
17
18 print("\nEigenvalues of B:")
19 for i, eigenvalue in enumerate(eigenvalues_B):
20     print(f"Eigenvalue {i+1}: {eigenvalue:.4f}")
21
22 # Calculate the sum of eigenvalues
23 sum_of_eigenvalues_B = np.sum(eigenvalues_B)
24
25 # Print the sum of eigenvalues and compare with trace
26 print(f"\nSum of Eigenvalues: {sum_of_eigenvalues_B:.4f}")
27 print(f"Trace of B: {trace_B}")
28
29 # Check if they are approximately equal
30 if np.isclose(sum_of_eigenvalues_B, trace_B):
31     print("\nThe sum of the eigenvalues is approximately equal to the trace of the matrix.")
32 else:
33     print("\nThe sum of the eigenvalues is not equal to the trace of the matrix.")
34
```

# Trace and Eigenvalues of a Non-Diagonal 3x3 Matrix

```
1 # Define a new 3x3 matrix with real eigenvalues
2 B = np.array([[4, 2, 1],
3               [2, 3, 1],
4               [1, 1, 2]])
5
6 # Calculate the trace of the new matrix
7 trace_B = np.trace(B)
8
9 # Calculate the eigenvalues of the new matrix
10 eigenvalues_B, _ = eig(B)
11
12 # Display results with better print formatting
13 print("Matrix B:")
14 print(B)
15 print("\nTrace of B (sum of diagonal elements):")
16 print(f"Trace(B) = {trace_B}")
17
18 print("\nEigenvalues of B:")
19 for i, eigenvalue in enumerate(eigenvalues_B):
20     print(f"Eigenvalue {i+1}: {eigenvalue:.4f}")
21
22 # Calculate the sum of eigenvalues
23 sum_of_eigenvalues_B = np.sum(eigenvalues_B)
24
25 # Print the sum of eigenvalues and compare with trace
26 print(f"\nSum of Eigenvalues: {sum_of_eigenvalues_B:.4f}")
27 print(f"Trace of B: {trace_B}")
28
29 # Check if they are approximately equal
30 if np.isclose(sum_of_eigenvalues_B, trace_B):
31     print("\nThe sum of the eigenvalues is approximately equal to the trace of the matrix.")
32 else:
33     print("\nThe sum of the eigenvalues is not equal to the trace of the matrix.")
34
```

# Trace and Eigenvalues of a Non-Diagonal 3x3 Matrix

Matrix  $A$ :

$$A = \begin{bmatrix} 2 & 3 & 1 \\ 1 & 5 & 4 \\ 3 & 2 & 6 \end{bmatrix}$$

Trace of  $A$  (sum of diagonal elements):

$$\text{Trace}(A) = 2 + 5 + 6 = 13$$

Eigenvalues of  $A$ :

- Eigenvalue 1:  $9.5618 + 0.0000j$
- Eigenvalue 2:  $1.7191 + 1.4728j$
- Eigenvalue 3:  $1.7191 - 1.4728j$

Sum of Eigenvalues:

$$\lambda_1 + \lambda_2 + \lambda_3 = 9.5618 + (1.7191 + 1.4728j) + (1.7191 - 1.4728j) = 13 + 0j$$

Conclusion:

- The sum of the eigenvalues  $13 + 0j$  is approximately equal to the trace of matrix  $A$ , which is 13.

Matrix A:

```
[[2 3 1]
 [1 5 4]
 [3 2 6]]
```

Trace of A (sum of diagonal elements):

Trace(A) = 13

Eigenvalues of A:

Eigenvalue 1: 9.5618+0.0000j

Eigenvalue 2: 1.7191+1.4728j

Eigenvalue 3: 1.7191-1.4728j

Sum of Eigenvalues: 13.0000+0.0000j

Trace of A: 13

The sum of the eigenvalues is approximately equal to the trace of the matrix.

# Determinant and Eigenvalues of a Matrix

## Definition: Determinant:

- The **determinant** of a square matrix  $A$  is a scalar value that describes certain properties of the matrix, such as:
  - Whether the matrix is invertible (non-zero determinant indicates invertibility).
  - The scaling factor of the transformation described by the matrix.
- For a  $n \times n$  matrix  $A$ , the determinant is denoted as  $\det(A)$ .

## Relationship to Eigenvalues:

- The determinant of a matrix is the **product of its eigenvalues**.

If  $\lambda_1, \lambda_2, \dots, \lambda_n$  are the eigenvalues of matrix  $A$ , then:

$$\det(A) = \lambda_1 \times \lambda_2 \times \dots \times \lambda_n$$

# Determinant and Eigenvalues of a Matrix

## Numerical Example:

- For matrix  $A$ :

$$A = \begin{bmatrix} 4 & 2 & 1 \\ 2 & 3 & 1 \\ 1 & 1 & 2 \end{bmatrix}$$

- Eigenvalues:  $\lambda_1 \approx 6.05$ ,  $\lambda_2 \approx 1.64$ ,  $\lambda_3 \approx 1.31$
- Determinant:

$$\det(A) = 6.05 \times 1.64 \times 1.31 \approx 13.00$$

## Conclusion:

- The determinant gives insight into the nature of the matrix transformation, and its value is directly linked to the eigenvalues by their product.

# Determinant and Eigenvalues of a Matrix

```
1 # Define the 3x3 matrix
2 C = np.array([[4, 2, 1],
3               [2, 3, 1],
4               [1, 1, 2]])
5
6 # Calculate the determinant of the matrix
7 det_C = np.linalg.det(C)
8
9 # Calculate the eigenvalues of the matrix
10 eigenvalues_C, _ = eig(C)
11
12 # Product of the eigenvalues
13 product_eigenvalues_C = np.prod(eigenvalues_C)
14
15 # Printing the results
16 print(f"Matrix C:\n{C}")
17 print(f"\nDeterminant of C: {det_C:.4f}")
18 print("\nEigenvalues of C:")
19 for i, eigenvalue in enumerate(eigenvalues_C, 1):
20     print(f"Eigenvalue {i}: {eigenvalue:.4f}")
21
22 print(f"\nProduct of Eigenvalues: {product_eigenvalues_C:.4f}")
23
24 # Check if determinant and product of eigenvalues match
25 if np.isclose(det_C, product_eigenvalues_C):
26     print("\nThe determinant is approximately equal to the product of the eigenvalues.")
27 else:
28     print("\nThe determinant is not equal to the product of the eigenvalues.")
29
```

Matrix C:

```
[[4 2 1]
 [2 3 1]
 [1 1 2]]
```

Determinant of C: 13.0000

Eigenvalues of C:

Eigenvalue 1: 6.0489

Eigenvalue 2: 1.6431

Eigenvalue 3: 1.3080

Product of Eigenvalues: 13.0000

The determinant is approximately equal to the product of the eigenvalues.

# Properties of Eigenvalues and Eigenvectors

```
1  # Importing necessary libraries
2  import numpy as np
3
4  # Define a matrix A (3x3 for illustration)
5  A = np.array([[4, 1, 2],
6               [1, 3, 1],
7               [2, 1, 3]])
8
9  # Compute the eigenvalues of matrix A
10 eigenvalues = np.linalg.eigvals(A)
11
12 # Compute the trace of A (sum of diagonal elements)
13 trace_A = np.trace(A)
14
15 # Compute the determinant of A
16 det_A = np.linalg.det(A)
17
18 # Compute the sum and product of eigenvalues
19 sum_eigenvalues = np.sum(eigenvalues)
20 product_eigenvalues = np.prod(eigenvalues)
21
```

Matrix A:

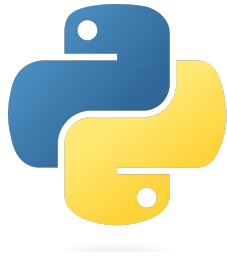
```
[[4 1 2]
 [1 3 1]
 [2 1 3]]
```

Trace of A (sum of diagonal elements): 10.00

Sum of eigenvalues: 10.00

Determinant of A: 21.00

Product of eigenvalues: 21.00



## Module 7: EigenValues and EigenVectors

### 7.2 Properties of Matrices with Eigenvalues and Eigenvectors



## CS316 INTRODUCTION TO AI AND DATA SCIENCE

### CHAPTER 3 PANDAS AND NUMPY

#### LECTURE 1 *PANDAS*

# Properties of Matrices with Eigenvalues and Eigenvectors

## Key Properties:

### 1. Square Matrix:

- The matrix must be **square** (i.e.,  $n \times n$ ) to have eigenvalues and eigenvectors.
- Example: A  $3 \times 3$  matrix like:

$$A = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

### 2. Non-Singular Matrix (for non-zero eigenvalues):

- A matrix is **non-singular** if its determinant is non-zero ( $\det(A) \neq 0$ ).
- If the determinant is zero, at least one eigenvalue will be zero, indicating a singular matrix.

### 3. Real or Complex Eigenvalues:

- Eigenvalues can be either **real** or **complex**.
- Real symmetric matrices have real eigenvalues, while other matrices may have complex eigenvalues.

# Properties of Matrices with Eigenvalues and Eigenvectors

## 4. Diagonalizable Matrices:

- A matrix is **diagonalizable** if it can be written as  $A = PDP^{-1}$ , where  $P$  is the matrix of eigenvectors and  $D$  is the diagonal matrix of eigenvalues.
- Not all matrices are diagonalizable, but those with distinct eigenvalues always are.

## 5. Eigenvalue Multiplicity:

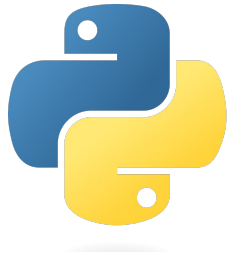
- **Algebraic Multiplicity:** The number of times an eigenvalue appears as a root of the characteristic equation.
- **Geometric Multiplicity:** The number of linearly independent eigenvectors corresponding to an eigenvalue.

## Examples:

- For matrix  $A = \begin{bmatrix} 4 & 2 & 1 \\ 2 & 3 & 1 \\ 1 & 1 & 2 \end{bmatrix}$ :
  - It is square, non-singular, and has real eigenvalues and eigenvectors.

## Conclusion:

- Only square matrices have eigenvalues and eigenvectors, and their properties depend on the matrix's structure (e.g., singularity, symmetry).



## Module 7: EigenValues and EigenVectors

### 7.2 How to Find Eigenvalues and Eigenvectors



## **CS316** INTRODUCTION TO AI AND DATA SCIENCE

### CHAPTER 3 PANDAS AND NUMPY

#### LECTURE 1 *PANDAS*

# Finding Eigenvalues

## Step 1: Finding Eigenvalues ( $\lambda$ )

### 1. Set up the Characteristic Equation:

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

- $\det$  represents the **determinant**.
- $I$  is the **identity matrix** of the same dimension as matrix  $A$ .

### 2. Solve for $\lambda$ :

- The resulting equation will be a polynomial in  $\lambda$ .
- The solutions to this polynomial are the **eigenvalues** of  $A$ .

# Finding Eigenvalues

**Example:**

For a matrix  $A = \begin{bmatrix} 4 & 2 \\ 1 & 3 \end{bmatrix}$ :

1. Set up the characteristic equation:

$$\det(A - \lambda I) = \det \begin{bmatrix} 4 - \lambda & 2 \\ 1 & 3 - \lambda \end{bmatrix} = 0$$

2. Solve for eigenvalues  $\lambda$ .
3. For each eigenvalue, solve  $(A - \lambda I)v = 0$  to find the eigenvectors.

# Step 1: Finding Eigenvalues

1. Set up the characteristic equation:

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

Substituting  $A$  and  $I$ :

$$\det \begin{bmatrix} 4 - \lambda & 2 \\ 1 & 3 - \lambda \end{bmatrix} = 0$$

2. Calculate the determinant:

$$(4 - \lambda)(3 - \lambda) - (2)(1) = 0$$

Expanding the determinant:

$$\lambda^2 - 7\lambda + 10 = 0$$

3. Solve for  $\lambda$ : Solving the quadratic equation:

$$\lambda_1 = 2, \quad \lambda_2 = 5$$

The eigenvalues of matrix  $A$  are  $\lambda_1 = 2$  and  $\lambda_2 = 5$ .

```
1  # Define the 2x2 matrix
2  A_example = np.array([[4, 2],
3  | | | | | | | | [1, 3]])
4
5  # Identity matrix
6  I_example = np.eye(2)
7
8  # Symbolic eigenvalue variable (lambda)
9  from sympy import symbols, Eq, det, Matrix
10
11 # Define lambda symbolically
12 lambda_sym = symbols('lambda')
13
14 # Define the matrix A - lambda*I
15 A_lambda = Matrix(A_example) - lambda_sym * Matrix(I_example)
16
17 # Calculate the determinant to find the characteristic equation
18 char_eq = det(A_lambda)
19
20 # Solve for the eigenvalues (lambda)
21 from sympy import solve
22 eigenvalues_example = solve(char_eq, lambda_sym)
23
24 eigenvalues_example
25
```

[2.000000000000000, 5.000000000000000]

# Step 2: Finding Eigenvectors

## Step 2: Finding Eigenvectors ( $v$ )

Now, for each eigenvalue, we will solve  $(A - \lambda I)v = 0$  to find the corresponding eigenvectors.

1. For  $\lambda_1 = 2$ :

$$(A - 2I)v = \begin{bmatrix} 4-2 & 2 \\ 1 & 3-2 \end{bmatrix} = \begin{bmatrix} 2 & 2 \\ 1 & 1 \end{bmatrix}$$

Solving  $\begin{bmatrix} 2 & 2 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$ , we get:

$$x = -y$$

So, the eigenvector corresponding to  $\lambda_1 = 2$  is:

$$v_1 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

2. For  $\lambda_2 = 5$ :

$$(A - 5I)v = \begin{bmatrix} 4-5 & 2 \\ 1 & 3-5 \end{bmatrix} = \begin{bmatrix} -1 & 2 \\ 1 & -2 \end{bmatrix}$$

Solving  $\begin{bmatrix} -1 & 2 \\ 1 & -2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$ , we get:

$$x = 2y$$

So, the eigenvector corresponding to  $\lambda_2 = 5$  is:

$$v_2 = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

## Final Result:

- Eigenvalues:

- $\lambda_1 = 2$

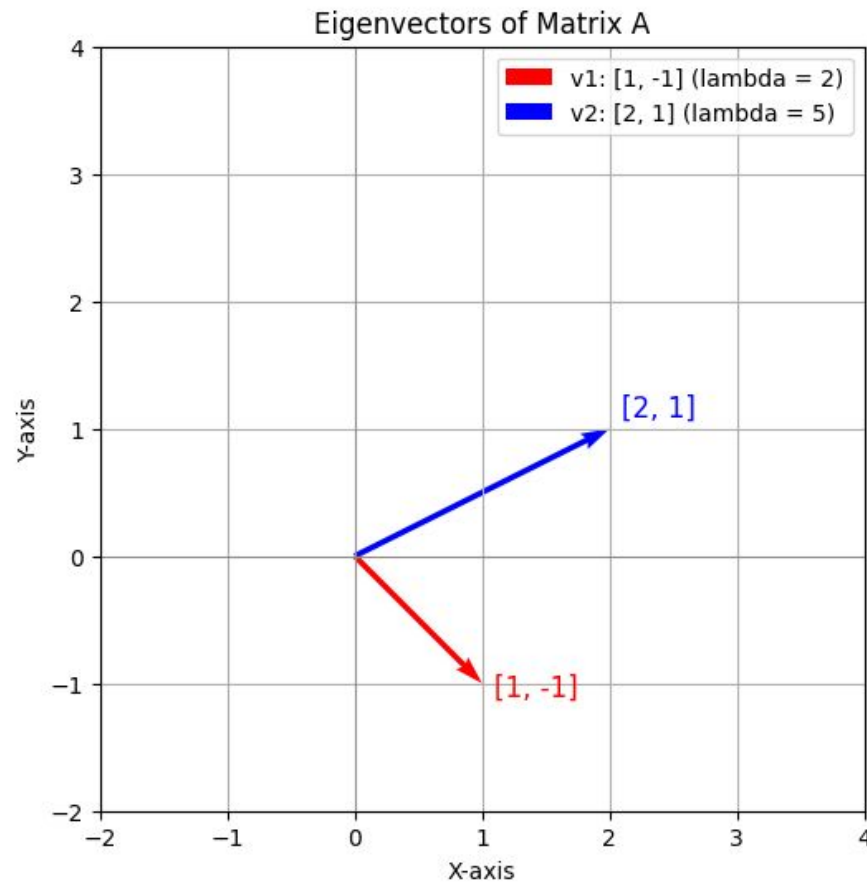
- $\lambda_2 = 5$

- Eigenvectors:

- For  $\lambda_1 = 2$ :  $v_1 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$

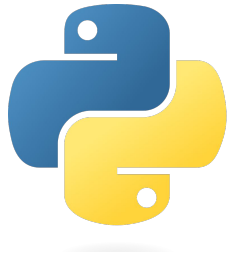
- For  $\lambda_2 = 5$ :  $v_2 = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$

# Visualization of the Eigenvectors



## Final Result:

- Eigenvalues:
  - $\lambda_1 = 2$
  - $\lambda_2 = 5$
- Eigenvectors:
  - For  $\lambda_1 = 2$ :  $v_1 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$
  - For  $\lambda_2 = 5$ :  $v_2 = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$



## Module 7: EigenValues and EigenVectors

### 7.3 Principal Component Analysis (PCA) A Use Case in Data Science



## CS316 INTRODUCTION TO AI AND DATA SCIENCE

### CHAPTER 3 PANDAS AND NUMPY

#### LECTURE 1 *PANDAS*

# Motivating Example – Why Use PCA?

- Imagine you're analyzing a dataset of students' performance across multiple subjects (Math, Physics, Chemistry, etc.).
- Challenge:** The dataset has many variables, and we suspect that some subjects are correlated (e.g., students who do well in Math may also do well in Physics).
- Goal:** Simplify the dataset by reducing the number of subjects, while retaining the most important information about student performance.

## Dataset:

- 10 students with scores in 4 subjects.
- Our goal is to simplify the analysis of student performance by reducing the number of features (subjects) while keeping the most important information.

| Student | Math | Physics | Chemistry | English |
|---------|------|---------|-----------|---------|
| 1       | 85   | 82      | 88        | 79      |
| 2       | 78   | 75      | 79        | 74      |
| 3       | 92   | 90      | 94        | 85      |
| ...     | ...  | ...     | ...       | ...     |
| 10      | 87   | 86      | 89        | 82      |

```
data = {  
    'Math': [85, 78, 92, 88, 76, 95, 89, 84, 91, 87],  
    'Physics': [82, 75, 90, 85, 73, 94, 88, 81, 89, 86],  
    'Chemistry': [88, 79, 94, 91, 77, 97, 90, 86, 92, 89],  
    'English': [79, 74, 85, 80, 70, 88, 83, 77, 84, 82]  
}
```

# Step 1 – Standardize the Data (Normalization)

Equation:

$$Z = \frac{X - \mu}{\sigma}$$

Where:

- $X$  is the original data.
- $\mu$  is the mean.
- $\sigma$  is the standard deviation.

```
# Convert the data into a DataFrame
df = pd.DataFrame(data)

# Step 1: Standardize the data
# Subtract the mean and divide by standard deviation to standardize
df_standardized = (df - df.mean()) / df.std()
```

Numerical Example (Math):

$$Z_{\text{Math, Student 1}} = \frac{85 - 87.7}{5.49} = -0.49$$

This step ensures that each subject contributes equally to the analysis.

# Step 1 – Standardize the Data (Normalization)

| Student | Math  | Physics | Chemistry | English |
|---------|-------|---------|-----------|---------|
| 1       | -0.25 | -0.35   | -0.05     | -0.22   |
| 2       | -1.42 | -1.40   | -1.49     | -1.15   |
| 3       | 0.92  | 0.86    | 0.91      | 0.89    |
| 4       | 0.25  | 0.11    | 0.43      | -0.04   |
| 5       | -1.75 | -1.70   | -1.81     | -1.88   |
| 6       | 1.42  | 1.46    | 1.39      | 1.44    |
| 7       | 0.42  | 0.56    | 0.27      | 0.52    |
| 8       | -0.42 | -0.50   | -0.37     | -0.59   |
| 9       | 0.75  | 0.71    | 0.59      | 0.70    |
| 10      | 0.08  | 0.26    | 0.11      | 0.33    |

## Step 2 – Calculate the Covariance Matrix

What is the Covariance Matrix?

- It shows how much each feature (subject) varies with others.

Covariance Matrix:

$$\text{Cov} = \begin{bmatrix} \text{Var}(\text{Math}) & \text{Cov}(\text{Math}, \text{Physics}) & \dots \\ \text{Cov}(\text{Physics}, \text{Math}) & \text{Var}(\text{Physics}) & \dots \\ \vdots & \vdots & \ddots \end{bmatrix}$$

Example:

- Math and Physics might have high covariance because they are closely related.

## Step 2 – Covariance Matrix and Eigen Decomposition

**Covariance Matrix:** The covariance matrix represents how the variables (subjects) vary together.

**Equation for Covariance:**

$$\text{Cov}(X, Y) = \frac{1}{n-1} \sum_{i=1}^n (X_i - \bar{X})(Y_i - \bar{Y})$$

**Eigenvalues and Eigenvectors:**

- **Eigenvalues** tell us the amount of variance explained by each principal component.
- **Eigenvectors** define the direction of each principal component.

## Step 2 – Calculate the Covariance Matrix

```
### Step 2: Calculate the Covariance Matrix
cov_matrix = np.cov(df_standardized.T)
print("\nCovariance Matrix:\n", cov_matrix)
```

Covariance Matrix:

```
[[1.          0.99465467 0.99265481 0.98433819]
 [0.99465467 1.          0.98031195 0.99155631]
 [0.99265481 0.98031195 1.          0.96962717]
 [0.98433819 0.99155631 0.96962717 1.          ]]
```

$$\text{Covariance Matrix} = \begin{bmatrix} 1.00 & 0.99 & 0.99 & 0.98 \\ 0.99 & 1.00 & 0.98 & 0.99 \\ 0.99 & 0.98 & 1.00 & 0.97 \\ 0.98 & 0.99 & 0.97 & 1.00 \end{bmatrix}$$

# Step 3 – Eigen Decomposition

- Eigenvalues:

Eigenvalues = [3.9566, 0.0338, 0.0079, 0.0016]

- Eigenvectors:

$$\begin{bmatrix} 0.5019 & 0.2018 & -0.3465 & -0.7664 \\ 0.5013 & -0.2614 & -0.6227 & 0.5409 \\ 0.4982 & 0.6969 & 0.3959 & 0.3308 \\ 0.4986 & -0.6366 & 0.5792 & -0.1029 \end{bmatrix}$$

## Interpretation:

- The first eigenvalue (**3.9566**) explains the largest portion of the variance, and the first eigenvector points in the direction of maximum variation.

```
### Step 3: Compute Eigenvalues and Eigenvectors of the Covariance Matrix
eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)
```

```
print("\nEigenvalues:\n", eigenvalues)
print("\nEigenvectors:\n", eigenvectors)
```

Eigenvalues:

```
[3.95661230e+00 3.38356070e-02 1.65383113e-03 7.89826523e-03]
```

Eigenvectors:

```
[[ 0.50190456  0.20177656 -0.76638646 -0.34645321]
 [ 0.50125904 -0.26142512  0.54094842 -0.62271268]
 [ 0.49822683  0.69686993  0.33080625  0.39586558]
 [ 0.49859925 -0.63664388 -0.1029263   0.57921458]]
```

# Step4 – Sorting Eigenvalues and Eigenvectors

**Goal:** Arrange the eigenvalues and their corresponding eigenvectors in descending order of importance (variance explained).

## Numerical Results:

- **Eigenvalues** (sorted in descending order):

$$\lambda_1 = 3.9566, \quad \lambda_2 = 0.0338, \quad \lambda_3 = 0.0079, \quad \lambda_4 = 0.0016$$

- **Eigenvectors** (sorted based on the eigenvalues):

$$\text{Eigenvector}_1 = \begin{bmatrix} 0.5019 \\ 0.5013 \\ 0.4982 \\ 0.4986 \end{bmatrix}, \quad \text{Eigenvector}_2 = \begin{bmatrix} 0.2018 \\ -0.2614 \\ 0.6969 \\ -0.6366 \end{bmatrix}, \dots$$

## Interpretation:

- The first eigenvalue ( $\lambda_1 = 3.9566$ ) explains the majority of the variance in the data.
- Eigenvector 1 is the direction that captures the most variation in the data.

```
### Step 4: Sort Eigenvalues and Eigenvectors (i
# Sort the eigenvalues and their corresponding e
sorted_indices = np.argsort(eigenvalues)[::-1]
eigenvalues = eigenvalues[sorted_indices]
eigenvectors = eigenvectors[:, sorted_indices]
```

Eigenvalues:

```
[3.95661230e+00 3.38356070e-02 1.65383113e-03 7.89826523e-03]
```

Eigenvectors:

```
[[ 0.50190456  0.20177656 -0.76638646 -0.34645321]
 [ 0.50125904 -0.26142512  0.54094842 -0.62271268]
 [ 0.49822683  0.69686993  0.33080625  0.39586558]
 [ 0.49859925 -0.63664388 -0.1029263   0.57921458]]
```

# PCA Transformation

**Goal:** Project the standardized data onto the eigenvectors (principal components).

**Transformation Equation:**

$$Z_{PC} = X_{\text{standardized}} \cdot \text{Eigenvector}$$

**PCA Transformed Data:**

| Student | PC1    | PC2    | PC3    | PC4    |
|---------|--------|--------|--------|--------|
| 1       | -0.434 | 0.148  | 0.155  | 0.011  |
| 2       | -2.727 | -0.227 | 0.113  | -0.044 |
| 3       | 1.788  | 0.031  | 0.021  | -0.029 |
| 4       | 0.375  | 0.347  | -0.003 | 0.012  |
| ...     | ...    | ...    | ...    | ...    |
| 10      | 0.392  | -0.184 | 0.048  | 0.077  |

**Interpretation:**

- The dataset has been transformed into the principal component space, where each student is now represented by principal components (PC1, PC2, PC3, PC4).

# PCA Transformation

**Goal:** Project standardized data onto principal components (eigenvectors).

---

**Compact Transformation Equation:**

$$Z_{PC} = X_{\text{standardized}} \cdot \text{Eigenvector}$$

Where:

- $X_{\text{standardized}}$  is the matrix of standardized data.
- $\text{Eigenvector}$  is the matrix of eigenvectors.
- $Z_{PC}$  is the transformed data in principal component space.

# PCA Transformation

**Goal: Project standardized data onto principal components (eigenvectors).**

**Compact Transformation Equation:**

$$Z_{PC} = X_{\text{standardized}} \cdot \text{Eigenvector}$$

Where:

- $X_{\text{standardized}}$  is the matrix of standardized data.
  - $\text{Eigenvector}$  is the matrix of eigenvectors.
  - $Z_{PC}$  is the transformed data in principal component space.
- 

**Expanded Equation for PC1:**

For each student (row in the dataset), the principal component  $Z_{PC1}$  is computed as:

$$Z_{PC1} = (X_1 \times e_{11}) + (X_2 \times e_{12}) + (X_3 \times e_{13}) + (X_4 \times e_{14})$$

Where:

- $X_1, X_2, X_3, X_4$  are the standardized values for Math, Physics, Chemistry, and English.
- $e_{11}, e_{12}, e_{13}, e_{14}$  are the elements of the first eigenvector.

# Numerical Application (Student 1)

- Standardized data for Student 1:

| Student | Math  | Physics | Chemistry | English |
|---------|-------|---------|-----------|---------|
| 1       | -0.25 | -0.35   | -0.05     | -0.22   |

$$X = [-0.25 \quad -0.35 \quad -0.05 \quad -0.22]$$

- First Eigenvector:

$$\text{Eigenvector}_1 = \begin{bmatrix} 0.5019 \\ 0.5013 \\ 0.4982 \\ 0.4986 \end{bmatrix}$$

- Computation of  $Z_{PC1}$ :

$$Z_{PC1} = (-0.25 \times 0.5019) + (-0.35 \times 0.5013) + (-0.05 \times 0.4982) + (-0.22 \times 0.4986)$$

$$Z_{PC1} = -0.1255 + -0.1755 + -0.0249 + -0.1097 = -0.434$$

# Numerical Application (Student 1)

| Student | Math  | Physics | Chemistry | English |
|---------|-------|---------|-----------|---------|
| 1       | -0.25 | -0.35   | -0.05     | -0.22   |

## Numerical Application for PC2 (Student 1):

- Second Eigenvector:

$$\text{Eigenvector}_2 = \begin{bmatrix} 0.2018 \\ -0.2614 \\ 0.6969 \\ -0.6366 \end{bmatrix}$$

- Computation of  $Z_{PC2}$ :

$$Z_{PC2} = (-0.25 \times 0.2018) + (-0.35 \times -0.2614) + (-0.05 \times 0.6969) + (-0.22 \times -0.6366)$$

$$Z_{PC2} = -0.0505 + 0.0915 + -0.0348 + 0.1401 = 0.147$$

# Numerical Application (Student 1)

## Transformed Data for Student 1:

- PC1:  $-0.434$
  - PC2:  $0.147$
- 

## Summary:

- PC1 represents the direction with the highest variance (captured by Eigenvector 1).
- PC2 adds more nuanced variation (captured by Eigenvector 2).
- This transformation reduces data dimensionality while retaining essential patterns.



# CS316 INTRODUCTION TO AI AND DATA SCIENCE

## CHAPTER 4 Integrating Linear Algebra with NumPy: Practical Foundations for Data Science

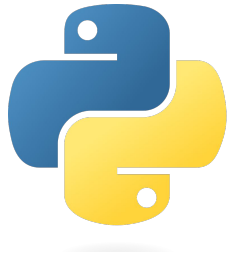
### LECTURE 3 *From Vector Projections to Principal Component Analysis: Unveiling the Geometry of Data in Feature Space*

Prof. Anis Koubaa

October 2024

# Questions

- What is a projection of a vector on another vector in space?
- Why covariance is used to select features?
- Why features projected on principal components represent the original data?



## Module 8: Projection in Space

### 8.1 Introduction to Projection in Space



## CS316 INTRODUCTION TO AI AND DATA SCIENCE

### CHAPTER 3 PANDAS AND NUMPY

#### LECTURE 1 *PANDAS*

# Projection of Vector onto the X-axis

- Dot Product ( $\mathbf{v} \cdot \mathbf{u}$ ):

$$[3, 2] \cdot [1, 0] = (3 \times 1) + (2 \times 0) = 3$$

- Norm of  $\mathbf{u}$  ( $\|\mathbf{u}\|$ ):

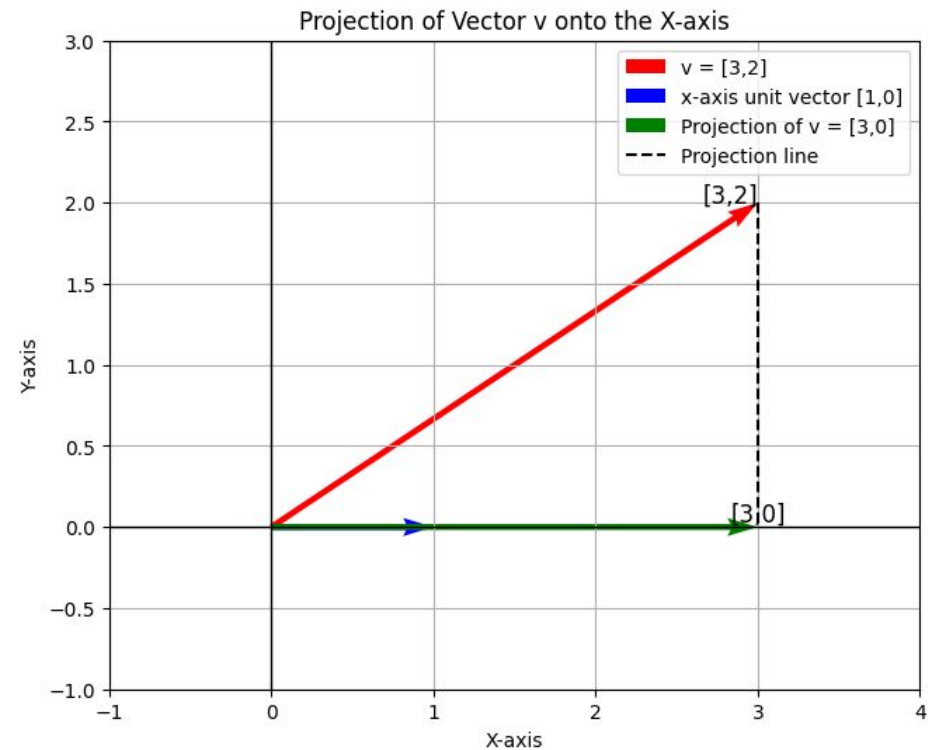
$$\|\mathbf{u}\| = \sqrt{1^2 + 0^2} = 1$$

- Projection Calculation:

$$\text{Proj}_{\mathbf{u}}(\mathbf{v}) = \frac{3}{1}[1, 0] = [3, 0]$$

## Numerical Example Summary:

- The dot product  $\mathbf{v} \cdot \mathbf{u} = 3$  measures the extent of  $\mathbf{v}$ 's alignment along the x-direction.
- The projection formula confirms that all of  $\mathbf{v}$ 's influence in the x-direction is captured, resulting in a vector along the x-axis.



# Projection of Vector $\mathbf{v}$ onto the Y-axis

## Dot Product and Norm Calculations:

- Dot Product ( $\mathbf{v} \cdot \mathbf{u}$ ):

$$[3, 2] \cdot [0, 1] = (3 \times 0) + (2 \times 1) = 2$$

- Norm of  $\mathbf{u}$  ( $\|\mathbf{u}\|$ ):

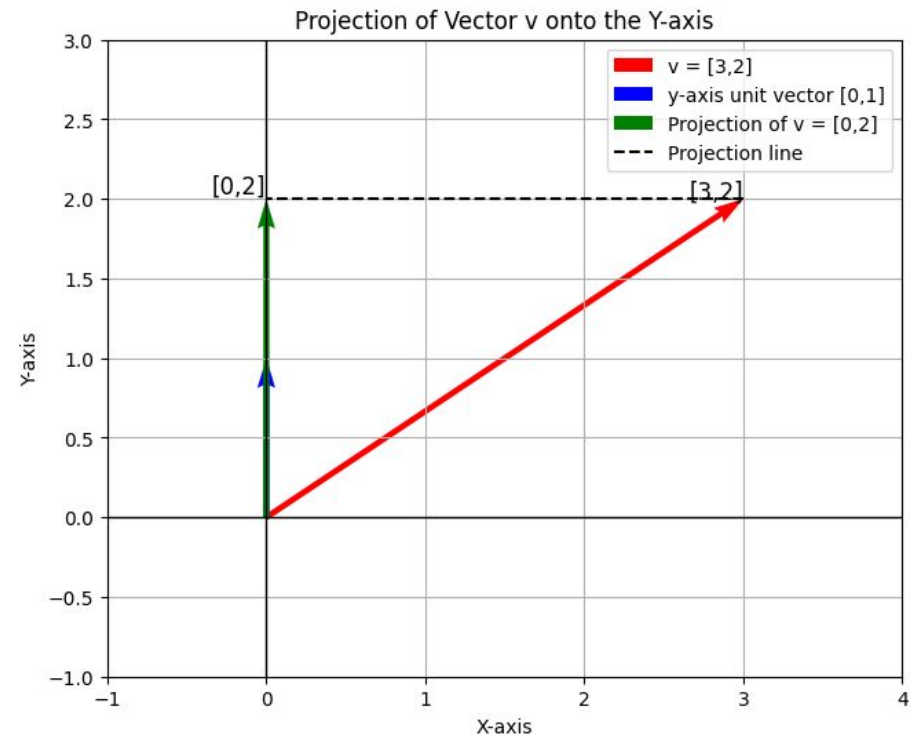
$$\|\mathbf{u}\| = \sqrt{0^2 + 1^2} = 1$$

- Projection Calculation:

$$\text{Proj}_{\mathbf{u}}(\mathbf{v}) = \frac{2}{1}[0, 1] = [0, 2]$$

## Numerical Example Summary:

- The dot product  $\mathbf{v} \cdot \mathbf{u} = 2$  captures the component of  $\mathbf{v}$  along the y-direction.
- The normalization ensures the projection scale is accurate, representing the y-component of  $\mathbf{v}$  without any scaling distortion.



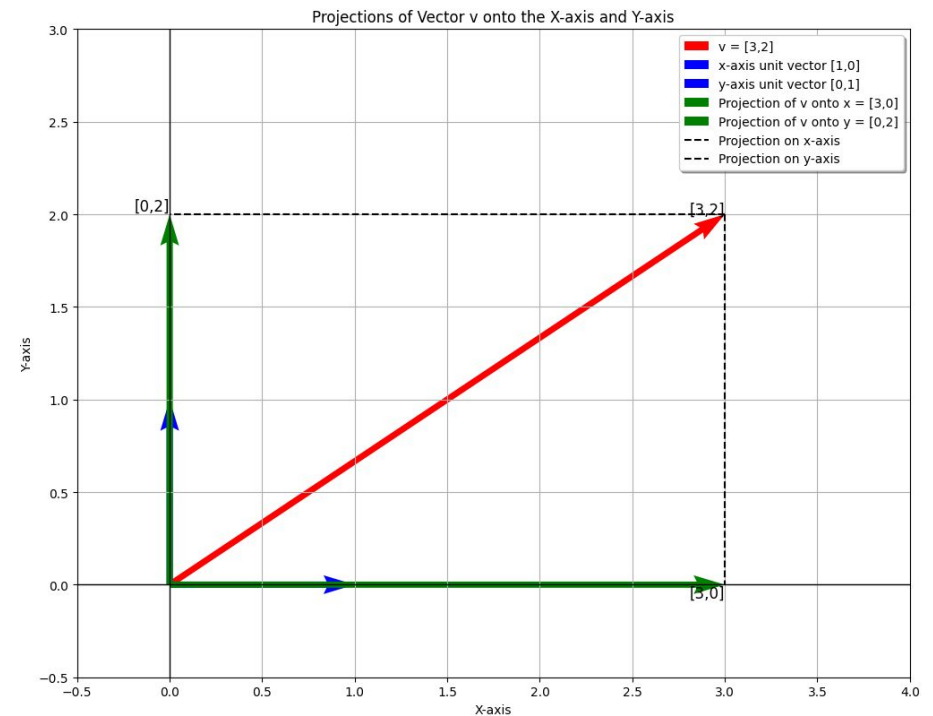
# Projection of Vector $\mathbf{v}$ onto the Y-axis

- Compact Formula:

$$\text{Proj}_{\mathbf{u}}(\mathbf{v}) = \frac{\mathbf{v} \cdot \mathbf{u}}{\|\mathbf{u}\|^2} \mathbf{u}$$

- Expanded Formula Using Norm:

$$\text{Proj}_{\mathbf{u}}(\mathbf{v}) = \left( \frac{\mathbf{v} \cdot \mathbf{u}}{\|\mathbf{u}\|^2} \right) \mathbf{u} = \left( \frac{\mathbf{v} \cdot \mathbf{u}}{\|\mathbf{u}\|} \right) \frac{\mathbf{u}}{\|\mathbf{u}\|}$$



# Projection of Vector $\mathbf{v}$ onto the Y-axis


- Compact Formula:

$$\text{Proj}_{\mathbf{u}}(\mathbf{v}) = \frac{\mathbf{v} \cdot \mathbf{u}}{\|\mathbf{u}\|^2} \mathbf{u}$$

- Expanded Formula Using Norm:

$$\text{Proj}_{\mathbf{u}}(\mathbf{v}) = \left( \frac{\mathbf{v} \cdot \mathbf{u}}{\|\mathbf{u}\|^2} \right) \mathbf{u} = \left( \frac{\mathbf{v} \cdot \mathbf{u}}{\|\mathbf{u}\|} \right) \frac{\mathbf{u}}{\|\mathbf{u}\|}$$

python

 Copy code

```
import numpy as np

# Define two vectors v and u
v = np.array([3, 4])
u = np.array([2, 1])

# 1. Projection of v onto u without normalization
proj_v_on_u = (np.dot(v, u) / np.dot(u, u)) * u

# 2. Projection of v onto the normalized vector u (with normalization)
u_norm = u / np.linalg.norm(u) # Normalized vector u
proj_v_on_u_normalized = np.dot(v, u_norm) * u_norm

# Results
print("Vector v:", v)
print("Vector u:", u)

print("\nProjection of v onto u (without normalization):", proj_v_on_u)
print("Projection of v onto u (with normalization):", proj_v_on_u_normalized)
```

# Introduction to Projection in Linear Algebra

## Objective:

- Understand the concept of projecting one vector onto another in a 2D space using principles of linear algebra.

## Definition of Projection:

- Projection** of vector  $\mathbf{v}$  onto vector  $\mathbf{u}$  involves finding the component of  $\mathbf{v}$  that lies along the direction of  $\mathbf{u}$ .

## Compact Formula:

- Projection of  $\mathbf{v}$  onto  $\mathbf{u}$ :

$$\text{Proj}_{\mathbf{u}}(\mathbf{v}) = \frac{\mathbf{v} \cdot \mathbf{u}}{\mathbf{u} \cdot \mathbf{u}} \mathbf{u}$$

## Expanded Formula Using Unit Vector:

- Unit vector  $\hat{\mathbf{u}}$  of  $\mathbf{u}$ :

$$\hat{\mathbf{u}} = \frac{\mathbf{u}}{\|\mathbf{u}\|}$$

- Projection using unit vector:

$$\text{Proj}_{\mathbf{u}}(\mathbf{v}) = (\mathbf{v} \cdot \hat{\mathbf{u}}) \hat{\mathbf{u}}$$

- Vectors:**

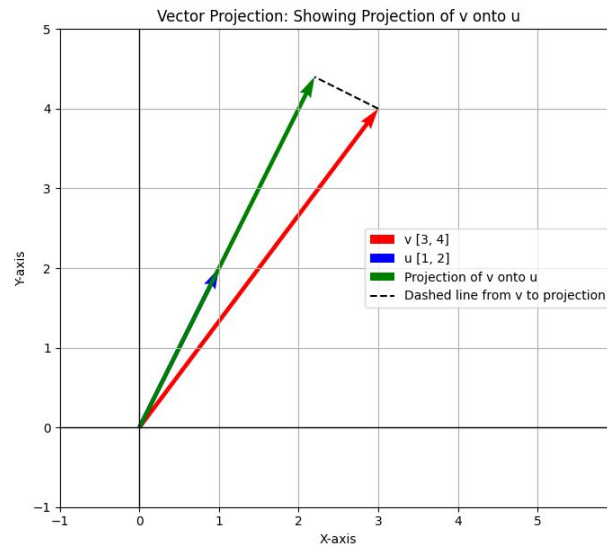
$$\mathbf{v} = \begin{bmatrix} 3 \\ 4 \end{bmatrix}, \quad \mathbf{u} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

- Unit Vector Calculation:**

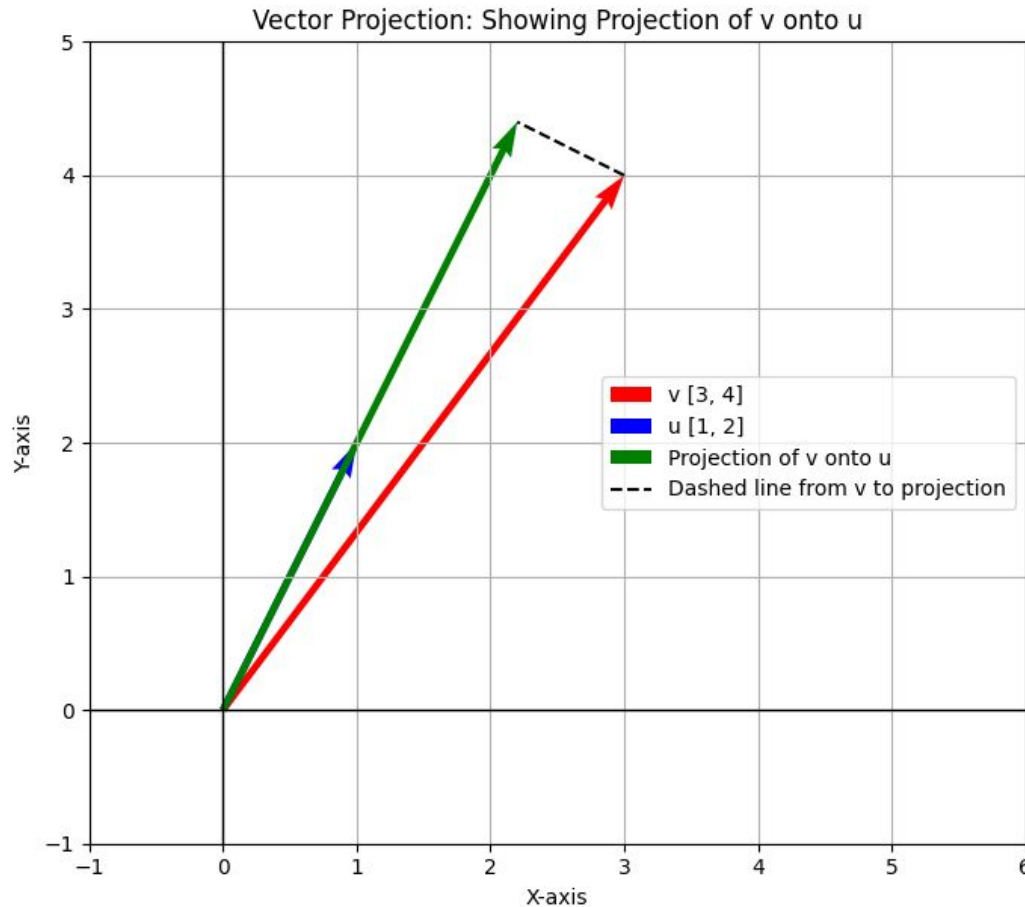
$$\hat{\mathbf{u}} = \frac{1}{\sqrt{5}} \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

- Projection Calculation:**

$$\begin{aligned} \text{Proj}_{\mathbf{u}}(\mathbf{v}) &= \left( \begin{bmatrix} 3 \\ 4 \end{bmatrix} \cdot \frac{1}{\sqrt{5}} \begin{bmatrix} 1 \\ 2 \end{bmatrix} \right) \frac{1}{\sqrt{5}} \begin{bmatrix} 1 \\ 2 \end{bmatrix} \\ &= \left( \frac{3 \times 1 + 4 \times 2}{5} \right) \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \frac{11}{5} \begin{bmatrix} 1 \\ 2 \end{bmatrix} \\ &= \begin{bmatrix} 2.2 \\ 4.4 \end{bmatrix} \end{aligned}$$



# Introduction to Projection in Linear Algebra



- Vectors:

$$\mathbf{v} = \begin{bmatrix} 3 \\ 4 \end{bmatrix}, \quad \mathbf{u} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

- Unit Vector Calculation:

$$\hat{\mathbf{u}} = \frac{1}{\sqrt{5}} \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

- Projection Calculation:

$$\begin{aligned} \text{Proj}_{\mathbf{u}}(\mathbf{v}) &= \left( \begin{bmatrix} 3 \\ 4 \end{bmatrix} \cdot \frac{1}{\sqrt{5}} \begin{bmatrix} 1 \\ 2 \end{bmatrix} \right) \frac{1}{\sqrt{5}} \begin{bmatrix} 1 \\ 2 \end{bmatrix} \\ &= \left( \frac{3 \times 1 + 4 \times 2}{5} \right) \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \frac{11}{5} \begin{bmatrix} 1 \\ 2 \end{bmatrix} \\ &= \begin{bmatrix} 2.2 \\ 4.4 \end{bmatrix} \end{aligned}$$

# Understanding the Role of Projection in Data Science

## Projection in Linear Algebra:

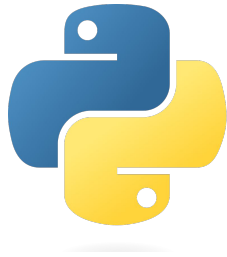
- **Basic Concept:** Projection is the process of mapping a vector onto a subspace, typically represented by another vector or a set of vectors (basis).
- **Mathematical Formulation:** Using the dot product and norms, projection quantifies how much one vector lies in the direction of another.

## Significance of Projection in Data Science:

- **Dimensionality Reduction:** In high-dimensional data analysis, projections are used to reduce the number of dimensions while preserving as much information as possible.
  - **Principal Component Analysis (PCA):** Projects high-dimensional data onto a lower-dimensional subspace where the variance is maximized.
  - **t-Distributed Stochastic Neighbor Embedding (t-SNE):** Projects complex data to two or three dimensions for visualization, focusing on maintaining local structures and relationships.

## Interpretation and Visualization:

- **Geometric Interpretation:** In visualization, projections help simplify complex datasets by reducing dimensions, allowing for easier exploration and understanding of data patterns.
- **Predictive Modeling:** Projections are used to transform features into a new space where they might be more effective predictors in machine learning models.



## Module 8: Projection in Space

### 8.2 Application of Projection in PCA.



## CS316 INTRODUCTION TO AI AND DATA SCIENCE

### CHAPTER 3 PANDAS AND NUMPY

#### LECTURE 1 *PANDAS*

# Reduce the Dimension of Features

| Student | Math | Physics | Chemistry | English |
|---------|------|---------|-----------|---------|
| 1       | 85   | 82      | 88        | 79      |
| 2       | 78   | 75      | 79        | 74      |
| 3       | 92   | 90      | 94        | 85      |
| ...     | ...  | ...     | ...       | ...     |
| 10      | 87   | 86      | 89        | 82      |

```
data = {  
    'Math': [85, 78, 92, 88, 76, 95, 89, 84, 91, 87],  
    'Physics': [82, 75, 90, 85, 73, 94, 88, 81, 89, 86],  
    'Chemistry': [88, 79, 94, 91, 77, 97, 90, 86, 92, 89],  
    'English': [79, 74, 85, 80, 70, 88, 83, 77, 84, 82]  
}
```

# Step 1 – Standardize the Data (Normalization)

| Student | Math  | Physics | Chemistry | English |
|---------|-------|---------|-----------|---------|
| 1       | -0.25 | -0.35   | -0.05     | -0.22   |
| 2       | -1.42 | -1.40   | -1.49     | -1.15   |
| 3       | 0.92  | 0.86    | 0.91      | 0.89    |
| 4       | 0.25  | 0.11    | 0.43      | -0.04   |
| 5       | -1.75 | -1.70   | -1.81     | -1.88   |
| 6       | 1.42  | 1.46    | 1.39      | 1.44    |
| 7       | 0.42  | 0.56    | 0.27      | 0.52    |
| 8       | -0.42 | -0.50   | -0.37     | -0.59   |
| 9       | 0.75  | 0.71    | 0.59      | 0.70    |
| 10      | 0.08  | 0.26    | 0.11      | 0.33    |

## Step 2 – Calculate the Covariance Matrix

```
### Step 2: Calculate the Covariance Matrix
cov_matrix = np.cov(df_standardized.T)
print("\nCovariance Matrix:\n", cov_matrix)
```

Covariance Matrix:

```
[[1.          0.99465467 0.99265481 0.98433819]
 [0.99465467 1.          0.98031195 0.99155631]
 [0.99265481 0.98031195 1.          0.96962717]
 [0.98433819 0.99155631 0.96962717 1.          ]]
```

$$\text{Covariance Matrix} = \begin{bmatrix} 1.00 & 0.99 & 0.99 & 0.98 \\ 0.99 & 1.00 & 0.98 & 0.99 \\ 0.99 & 0.98 & 1.00 & 0.97 \\ 0.98 & 0.99 & 0.97 & 1.00 \end{bmatrix}$$

# Importance of Eigenvalue Decomposition in Feature Reduction

## Objective:

- Understand why eigenvalue decomposition of the covariance matrix is essential for reducing the number of features in a dataset.

## Background:

- The **covariance matrix** quantifies the linear relationship between variables in a dataset. Each element represents the covariance between two features, reflecting how changes in one feature correspond with changes in another.

## Covariance Matrix Representation:

- The provided covariance matrix:

$$\begin{bmatrix} 1.00 & 0.99 & 0.99 & 0.98 \\ 0.99 & 1.00 & 0.98 & 0.99 \\ 0.99 & 0.98 & 1.00 & 0.97 \\ 0.98 & 0.99 & 0.97 & 1.00 \end{bmatrix}$$

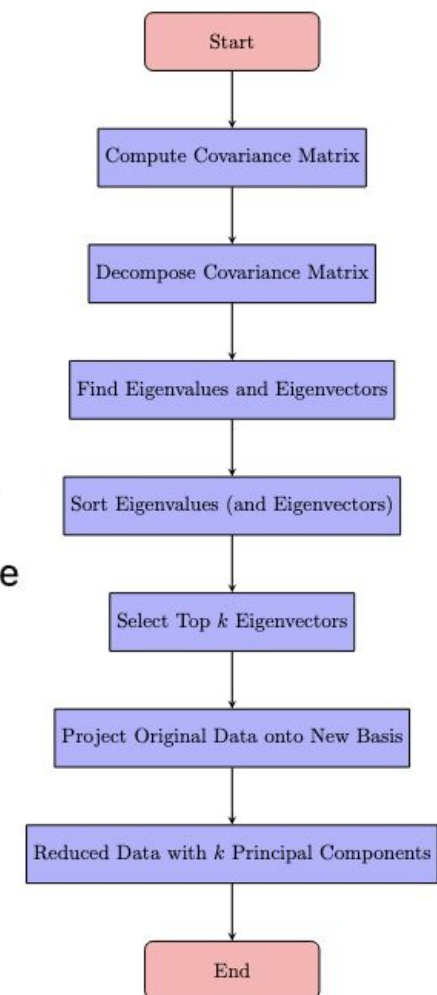
This matrix shows high correlations between all pairs of features, indicating redundant information.

# Importance of Eigenvalue Decomposition in Feature Reduction

- **Simplification:** Eigenvalue decomposition helps identify the principal components (directions in the feature space that maximize variance). This process transforms correlated features into a set of linearly uncorrelated features called principal components.
- **Dimensionality Reduction:**
  - **Eigenvalues:** Indicate the amount of variance carried by each principal component. A higher eigenvalue means more variance (information).
  - **Eigenvectors:** Define the direction of each principal component in the original feature space.

# Process of Reduction

1. **Decompose** the covariance matrix to find its eigenvalues and eigenvectors.
2. **Sort** the eigenvalues (and corresponding eigenvectors) from highest to lowest.
3. **Select** the top  $k$  eigenvectors as principal components to capture the significant variance.
4. **Project** the original data onto the space defined by the selected eigenvectors, reducing the dimensionality from four features to  $k$  principal components.



# Step 3 – Eigen Decomposition

- Eigenvalues:

Eigenvalues = [3.9566, 0.0338, 0.0079, 0.0016]

- Eigenvectors:

$$\begin{bmatrix} 0.5019 & 0.2018 & -0.3465 & -0.7664 \\ 0.5013 & -0.2614 & -0.6227 & 0.5409 \\ 0.4982 & 0.6969 & 0.3959 & 0.3308 \\ 0.4986 & -0.6366 & 0.5792 & -0.1029 \end{bmatrix}$$

## Interpretation:

- The first eigenvalue (3.9566) explains the largest portion of the variance, and the first eigenvector points in the direction of maximum variation.

```
### Step 3: Compute Eigenvalues and Eigenvectors of the Covariance Matrix
eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)
```

```
print("\nEigenvalues:\n", eigenvalues)
print("\nEigenvectors:\n", eigenvectors)
```

Eigenvalues:

```
[3.95661230e+00 3.38356070e-02 1.65383113e-03 7.89826523e-03]
```

Eigenvectors:

```
[[ 0.50190456  0.20177656 -0.76638646 -0.34645321]
 [ 0.50125904 -0.26142512  0.54094842 -0.62271268]
 [ 0.49822683  0.69686993  0.33080625  0.39586558]
 [ 0.49859925 -0.63664388 -0.1029263   0.57921458]]
```

# Step4 – Sorting Eigenvalues and Eigenvectors

**Goal:** Arrange the eigenvalues and their corresponding eigenvectors in descending order of importance (variance explained).

**Numerical Results:** **Dominant EigenVector and EigenValue**

- **Eigenvalues** (sorted in descending order):

$$\lambda_1 = 3.9566, \quad \lambda_2 = 0.0338, \quad \lambda_3 = 0.0079, \quad \lambda_4 = 0.0016$$

- **Eigenvectors** (sorted based on the eigenvalues):

$$\text{Eigenvector}_1 = \begin{bmatrix} 0.5019 \\ 0.5013 \\ 0.4982 \\ 0.4986 \end{bmatrix}, \quad \text{Eigenvector}_2 = \begin{bmatrix} 0.2018 \\ -0.2614 \\ 0.6969 \\ -0.6366 \end{bmatrix}, \dots$$

**Interpretation:**

- The first eigenvalue ( $\lambda_1 = 3.9566$ ) explains the majority of the variance in the data.
- Eigenvector 1 is the direction that captures the most variation in the data.

```
### Step 4: Sort Eigenvalues and Eigenvectors (i
# Sort the eigenvalues and their corresponding e
sorted_indices = np.argsort(eigenvalues)[::-1]
eigenvalues = eigenvalues[sorted_indices]
eigenvectors = eigenvectors[:, sorted_indices]
```

Eigenvalues:

```
[3.95661230e+00 3.38356070e-02 1.65383113e-03 7.89826523e-03]
```

Eigenvectors:

```
[[ 0.50190456  0.20177656 -0.76638646 -0.34645321]
 [ 0.50125904 -0.26142512  0.54094842 -0.62271268]
 [ 0.49822683  0.69686993  0.33080625  0.39586558]
 [ 0.49859925 -0.63664388 -0.1029263   0.57921458]]
```

# Transforming Data Using PCA – Case of Dominant Eigenvector


## Objective:

- Demonstrate how to transform high-dimensional standardized data into a new feature space using Principal Component Analysis (PCA).

## Dataset and Standardization:

- Standardized Data:

Features: Math, Physics, Chemistry, English



|         |         |         |         |
|---------|---------|---------|---------|
| -0.2506 | -0.3467 | -0.0480 | -0.2217 |
| -1.4200 | -1.4019 | -1.4869 | -1.1456 |
| 0.9188  | 0.8592  | 0.9113  | 0.8869  |
| 0.2506  | 0.1055  | 0.4317  | -0.0370 |
| -1.7541 | -1.7033 | -1.8066 | -1.8847 |
| 1.4200  | 1.4621  | 1.3909  | 1.4413  |
| 0.4176  | 0.5577  | 0.2718  | 0.5174  |
| -0.4176 | -0.4974 | -0.3677 | -0.5913 |
| 0.7517  | 0.7085  | 0.5915  | 0.7022  |
| 0.0835  | 0.2563  | 0.1119  | 0.3326  |

## First Eigenvector (Principal Component):

- Eigenvector 1:

[0.5019 0.5013 0.4982 0.4986]

## Transformation Formula:

- Projection Calculation:

$$Z_1 = \text{Data} \cdot \text{Eigenvector}_1$$

$$Z_1 = \begin{bmatrix} -0.2506 & -0.3467 & -0.0480 & -0.2217 \\ -1.4200 & -1.4019 & -1.4869 & -1.1456 \\ \dots & & & \\ 0.0835 & 0.2563 & 0.1119 & 0.3326 \end{bmatrix} \cdot \begin{bmatrix} 0.5019 \\ 0.5013 \\ 0.4982 \\ 0.4986 \end{bmatrix}$$

## Results of Projection:

- Projected Features (  $Z_1$  Values ):

$\begin{bmatrix} -0.4340 \\ -2.7274 \\ 1.7881 \\ 0.3753 \\ -3.5740 \\ 2.8572 \\ 0.8826 \\ -0.9369 \\ 1.3773 \\ 0.3920 \end{bmatrix}$

# Transforming Data Using PCA

## Interpretation and Benefits:



- **Dimensionality Reduction:** Reduces the dataset dimensions from four features to a single principal component.
- **Variance Capture:**  $Z_1$  captures the most variance across the dataset, enabling efficient data representation and analysis.

## Question

Is PCA Feature Representative of the Original Data?

| Math | Physics | Chemistry | English |
|------|---------|-----------|---------|
| 85   | 82      | 88        | 79      |
| 78   | 75      | 79        | 74      |
| 92   | 90      | 94        | 85      |
| 88   | 85      | 91        | 80      |
| 76   | 73      | 77        | 70      |
| 95   | 94      | 97        | 88      |
| 89   | 88      | 90        | 83      |
| 84   | 81      | 86        | 77      |
| 91   | 89      | 92        | 84      |
| 87   | 86      | 89        | 82      |

Features: Math, Physics, Chemistry, English


$$\begin{bmatrix} -0.2506 & -0.3467 & -0.0480 & -0.2217 \\ -1.4200 & -1.4019 & -1.4869 & -1.1456 \\ 0.9188 & 0.8592 & 0.9113 & 0.8869 \\ 0.2506 & 0.1055 & 0.4317 & -0.0370 \\ -1.7541 & -1.7033 & -1.8066 & -1.8847 \\ 1.4200 & 1.4621 & 1.3909 & 1.4413 \\ 0.4176 & 0.5577 & 0.2718 & 0.5174 \\ -0.4176 & -0.4974 & -0.3677 & -0.5913 \\ 0.7517 & 0.7085 & 0.5915 & 0.7022 \\ 0.0835 & 0.2563 & 0.1119 & 0.3326 \end{bmatrix}$$

$$\begin{bmatrix} -0.4340 \\ -2.7274 \\ 1.7881 \\ 0.3753 \\ -3.5740 \\ 2.8572 \\ 0.8826 \\ -0.9369 \\ 1.3773 \\ 0.3920 \end{bmatrix}$$
$$\begin{bmatrix} 1.00 & 0.99 & 0.99 & 0.98 \\ 0.99 & 1.00 & 0.98 & 0.99 \\ 0.99 & 0.98 & 1.00 & 0.97 \\ 0.98 & 0.99 & 0.97 & 1.00 \end{bmatrix}$$

# Validating PCA's Representation of Original Data

## Objective:

- Illustrate PCA's capability to maintain the essential characteristics of the original data through dimensionality reduction.

## Key Points:

### 1. Variance as Information:

- PCA preserves variance, which represents the most informative aspects of the data.

### 2. Principal Components:

- Identify directions that maximize data variance.
- Capture core patterns and structures in the data.

### 3. Mathematical Foundation:

- Covariance Matrix:** Identifies the covariance between all pairs of features.
- Eigenvalues and Eigenvectors:** Determine the amount of variance each component holds.

$$\begin{bmatrix} 1.00 & 0.99 & 0.99 & 0.98 \\ 0.99 & 1.00 & 0.98 & 0.99 \\ 0.99 & 0.98 & 1.00 & 0.97 \\ 0.98 & 0.99 & 0.97 & 1.00 \end{bmatrix}$$

Numerical Results:

#### Dominant EigenVector and EigenValue

- Eigenvalues** (sorted in descending order):

$$\lambda_1 = 3.9566, \quad \lambda_2 = 0.0338, \quad \lambda_3 = 0.0079, \quad \lambda_4 = 0.0016$$

- Eigenvectors** (sorted based on the eigenvalues):

$$\text{Eigenvector}_1 = \begin{bmatrix} 0.5019 \\ 0.5013 \\ 0.4982 \\ 0.4986 \end{bmatrix}, \quad \text{Eigenvector}_2 = \begin{bmatrix} 0.2018 \\ -0.2614 \\ 0.6969 \\ -0.6366 \end{bmatrix}, \dots$$

Interpretation:

- The first eigenvalue ( $\lambda_1 = 3.9566$ ) explains the majority of the variance in the data.
- Eigenvector 1 is the direction that captures the most variation in the data.

# Validating PCA's Representation of Original Data

$$Z_1 = [-1.50, -1.85, 0.97, 0.79, -1.48, 1.54, 0.59, -0.52, 0.91, 0.75]$$

## 4. Data Integrity:

- **Projection:** Transforms data to emphasize significant features without altering underlying relationships.
- **Explained Variance Ratio:** Quantifies the percentage of total variance represented by each principal component.

## 5. Practical Visualization:

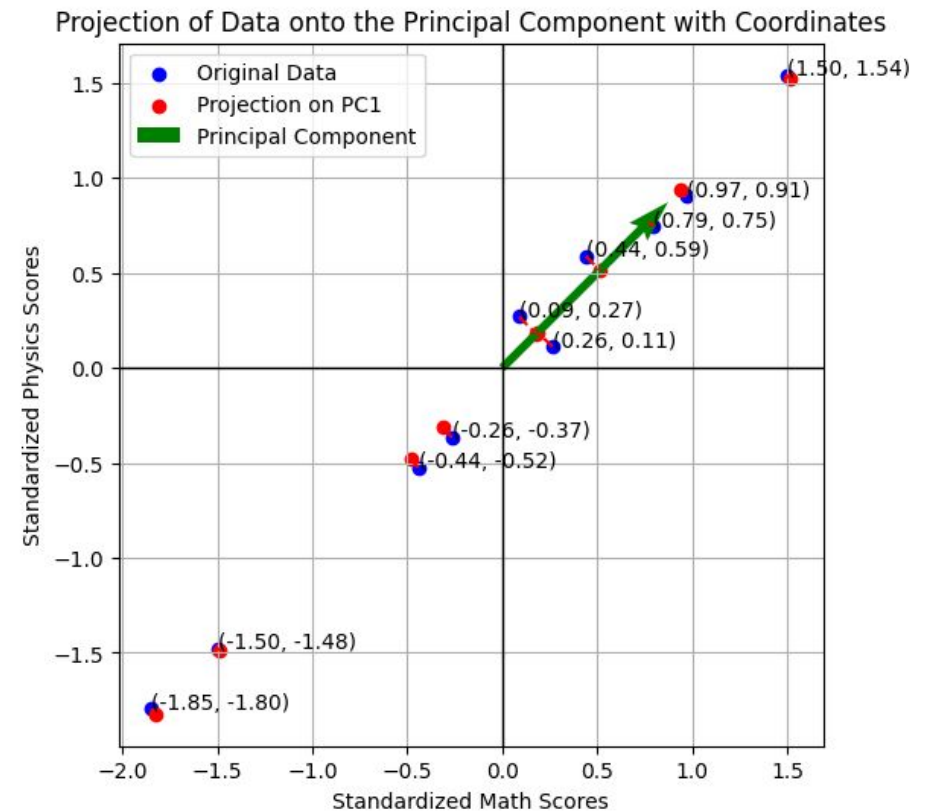
- Compare scatter plots before and after PCA to verify consistency in data trends and clusters.

## 6. Applications:

- **Face Recognition:** PCA in Eigenfaces shows effective feature retention.
- **Genomics:** Highlights patterns without distorting genetic information.

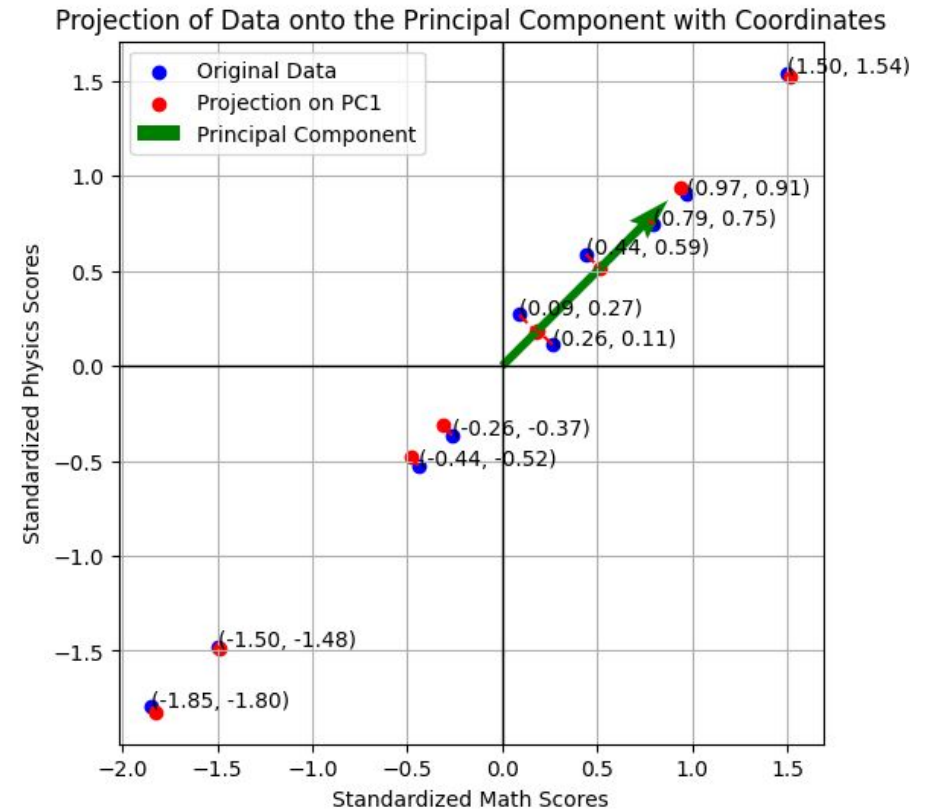
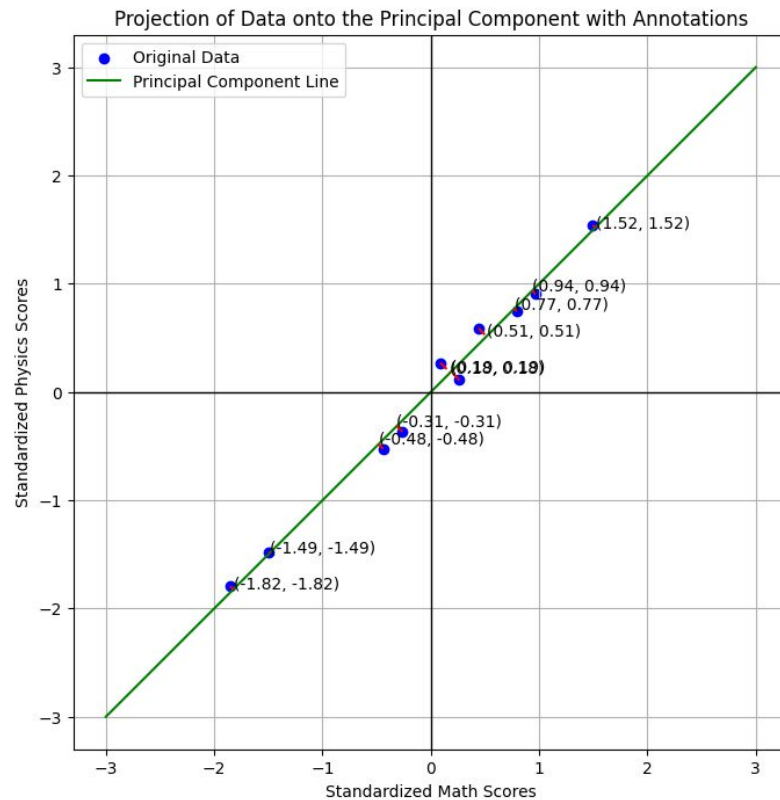
## 7. Conclusion:

- PCA ensures that dimension reduction does not compromise the data's fundamental characteristics, supporting its use in critical data analysis and feature reduction tasks.



# Validating PCA's Representation of Original Data

$$Z_1 = [-1.50, -1.85, 0.97, 0.79, -1.48, 1.54, 0.59, -0.52, 0.91, 0.75]$$



# PCA Transformation

**Goal:** Project the standardized data onto the eigenvectors (principal components).

**Transformation Equation:**

$$Z_{PC} = X_{\text{standardized}} \cdot \text{Eigenvector}$$

**PCA Transformed Data:**

| Student | PC1    | PC2    | PC3    | PC4    |
|---------|--------|--------|--------|--------|
| 1       | -0.434 | 0.148  | 0.155  | 0.011  |
| 2       | -2.727 | -0.227 | 0.113  | -0.044 |
| 3       | 1.788  | 0.031  | 0.021  | -0.029 |
| 4       | 0.375  | 0.347  | -0.003 | 0.012  |
| ...     | ...    | ...    | ...    | ...    |
| 10      | 0.392  | -0.184 | 0.048  | 0.077  |

**Interpretation:**

- The dataset has been transformed into the principal component space, where each student is now represented by principal components (PC1, PC2, PC3, PC4).

# PCA Transformation

**Goal:** Project standardized data onto principal components (eigenvectors).

---

**Compact Transformation Equation:**

$$Z_{PC} = X_{\text{standardized}} \cdot \text{Eigenvector}$$

Where:

- $X_{\text{standardized}}$  is the matrix of standardized data.
- $\text{Eigenvector}$  is the matrix of eigenvectors.
- $Z_{PC}$  is the transformed data in principal component space.

# PCA Transformation

**Goal: Project standardized data onto principal components (eigenvectors).**

**Compact Transformation Equation:**

$$Z_{PC} = X_{\text{standardized}} \cdot \text{Eigenvector}$$

Where:

- $X_{\text{standardized}}$  is the matrix of standardized data.
  - $\text{Eigenvector}$  is the matrix of eigenvectors.
  - $Z_{PC}$  is the transformed data in principal component space.
- 

**Expanded Equation for PC1:**

For each student (row in the dataset), the principal component  $Z_{PC1}$  is computed as:

$$Z_{PC1} = (X_1 \times e_{11}) + (X_2 \times e_{12}) + (X_3 \times e_{13}) + (X_4 \times e_{14})$$

Where:

- $X_1, X_2, X_3, X_4$  are the standardized values for Math, Physics, Chemistry, and English.
- $e_{11}, e_{12}, e_{13}, e_{14}$  are the elements of the first eigenvector.

# Numerical Application (Student 1)

- Standardized data for Student 1:

| Student | Math  | Physics | Chemistry | English |
|---------|-------|---------|-----------|---------|
| 1       | -0.25 | -0.35   | -0.05     | -0.22   |

$$X = [-0.25 \quad -0.35 \quad -0.05 \quad -0.22]$$

- First Eigenvector:

$$\text{Eigenvector}_1 = \begin{bmatrix} 0.5019 \\ 0.5013 \\ 0.4982 \\ 0.4986 \end{bmatrix}$$

- Computation of  $Z_{PC1}$ :

$$Z_{PC1} = (-0.25 \times 0.5019) + (-0.35 \times 0.5013) + (-0.05 \times 0.4982) + (-0.22 \times 0.4986)$$

$$Z_{PC1} = -0.1255 + -0.1755 + -0.0249 + -0.1097 = -0.434$$

# Numerical Application (Student 1)

| Student | Math  | Physics | Chemistry | English |
|---------|-------|---------|-----------|---------|
| 1       | -0.25 | -0.35   | -0.05     | -0.22   |

## Numerical Application for PC2 (Student 1):

- Second Eigenvector:

$$\text{Eigenvector}_2 = \begin{bmatrix} 0.2018 \\ -0.2614 \\ 0.6969 \\ -0.6366 \end{bmatrix}$$

- Computation of  $Z_{PC2}$ :

$$Z_{PC2} = (-0.25 \times 0.2018) + (-0.35 \times -0.2614) + (-0.05 \times 0.6969) + (-0.22 \times -0.6366)$$

$$Z_{PC2} = -0.0505 + 0.0915 + -0.0348 + 0.1401 = 0.147$$

# Numerical Application (Student 1)

## Transformed Data for Student 1:

- PC1:  $-0.434$
  - PC2:  $0.147$
- 

## Summary:

- PC1 represents the direction with the highest variance (captured by Eigenvector 1).
- PC2 adds more nuanced variation (captured by Eigenvector 2).
- This transformation reduces data dimensionality while retaining essential patterns.



# CS316 INTRODUCTION TO AI AND DATA SCIENCE

## CHAPTER 4 Integrating Linear Algebra with NumPy: Practical Foundations for Data Science

### LECTURE 4 *Singular Value Decomposition*

Prof. Anis Koubaa

October 2024

# Book Recommendation System

## What Is a Book Recommendation System?

- A system designed to predict and suggest books based on user preferences.

## Key Objectives:

1. **User Satisfaction:** Recommend books tailored to individual tastes.
2. **Platform Engagement:** Keep users engaged and active.
3. **Inventory Insight:** Guide inventory and marketing strategies.

## Challenges:

- **Diverse Preferences:** Wide range of user interests.
- **Sparse Data:** Users rate only a few books, creating gaps in the data.

## Need for SVD:

- **Address Sparsity and Diversity:** SVD helps uncover hidden patterns in sparse data, improving prediction accuracy.
- **Preparation for SVD:** Understand how decomposing the user-book matrix can enhance recommendation accuracy.

|     | Book 1 | Book 2 | Book 3 | Book 4 | Book 5 | Book 6 | Book 7 | Book 8 | Book 9 | Book 10 |
|-----|--------|--------|--------|--------|--------|--------|--------|--------|--------|---------|
| U1  | 0      | 5      | 3      | 0      | 0      | 0      | 0      | 4      | 1      | 3       |
| U2  | 0      | 5      | 4      | 0      | 0      | 0      | 0      | 0      | 0      | 0       |
| U3  | 1      | 0      | 0      | 0      | 0      | 3      | 0      | 0      | 0      | 0       |
| U4  | 1      | 0      | 0      | 5      | 5      | 4      | 0      | 0      | 2      | 0       |
| U5  | 0      | 0      | 0      | 5      | 0      | 2      | 0      | 0      | 0      | 0       |
| U6  | 5      | 3      | 5      | 4      | 0      | 5      | 0      | 0      | 0      | 0       |
| U7  | 0      | 0      | 4      | 0      | 0      | 0      | 0      | 4      | 0      | 5       |
| U8  | 3      | 0      | 0      | 4      | 3      | 3      | 3      | 0      | 0      | 0       |
| U9  | 4      | 1      | 0      | 0      | 0      | 0      | 3      | 1      | 4      | 0       |
| U10 | 0      | 3      | 3      | 0      | 3      | 0      | 0      | 0      | 0      | 0       |

# Motivating Example – Book Recommendation System

## Objective:

- **Improve Book Recommendations:** Utilize Singular Value Decomposition (SVD) to enhance the accuracy and personalization of book recommendations on digital reading platforms.

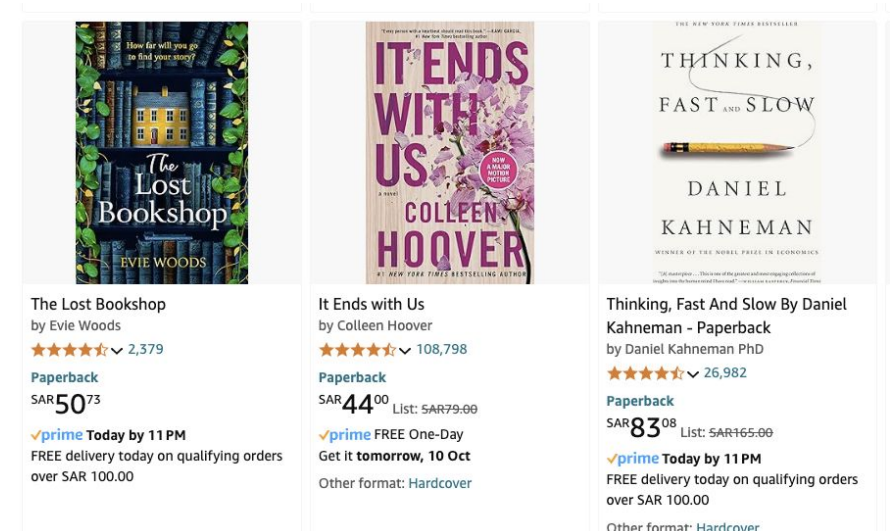
## Context and Problem Statement:

- **Growing Digital Libraries:** As digital libraries expand, users face overwhelming choices, making it challenging to discover books that match their interests.
- **User-Item Interaction Matrix:** Platforms collect data on user interactions (ratings, reviews, etc.) with books, forming a vast but sparse matrix due to users typically interacting with only a small selection of available books.

## Challenge:

- **Data Sparsity and Scalability:** Most of the user-item interaction matrix entries are empty (unrated books), complicating the prediction of user preferences.
- **Need for Personalization:** Enhancing user engagement through tailored recommendations, necessitating sophisticated algorithms to predict user preferences accurately.

| User \ Book | Book 1 | Book 2 | Book 3 | Book 4 | Book 5 |
|-------------|--------|--------|--------|--------|--------|
| User 1      | 4      | 0      | 0      | 5      | 3      |
| User 2      | 5      | 5      | 0      | 0      | 0      |
| User 3      | 0      | 2      | 3      | 4      | 0      |
| User 4      | 0      | 0      | 0      | 5      | 4      |
| User 5      | 3      | 4      | 0      | 0      | 0      |



# User-Item Rating Matrix for Book Recommendations

## Objective:

- Present the user-item rating matrix to illustrate the data structure used in book recommendation systems.

## User-Item Rating Matrix:

- This matrix represents the ratings given by 10 users to 10 different books:

|     | Book 1 | Book 2 | Book 3 | Book 4 | Book 5 | Book 6 | Book 7 | Book 8 | Book 9 | Book 10 |
|-----|--------|--------|--------|--------|--------|--------|--------|--------|--------|---------|
| U1  | 0      | 5      | 3      | 0      | 0      | 0      | 0      | 4      | 1      | 3       |
| U2  | 0      | 5      | 4      | 0      | 0      | 0      | 0      | 0      | 0      | 0       |
| U3  | 1      | 0      | 0      | 0      | 0      | 3      | 0      | 0      | 0      | 0       |
| U4  | 1      | 0      | 0      | 5      | 5      | 4      | 0      | 0      | 2      | 0       |
| U5  | 0      | 0      | 0      | 5      | 0      | 2      | 0      | 0      | 0      | 0       |
| U6  | 5      | 3      | 5      | 4      | 0      | 5      | 0      | 0      | 0      | 0       |
| U7  | 0      | 0      | 4      | 0      | 0      | 0      | 0      | 4      | 0      | 5       |
| U8  | 3      | 0      | 0      | 4      | 3      | 3      | 3      | 0      | 0      | 0       |
| U9  | 4      | 1      | 0      | 0      | 0      | 0      | 3      | 1      | 4      | 0       |
| U10 | 0      | 3      | 3      | 0      | 3      | 0      | 0      | 0      | 0      | 0       |

## Challenges in Building a Prediction Model:

### 1. Data Sparsity:

- The majority of the matrix entries are zeros, indicating that most books have not been rated by most users.
- Challenge: How to accurately predict these missing ratings?

### 2. Scalability:

- Handling potentially large datasets with thousands of users and books.
- Challenge: Efficient computation and storage solutions are needed to manage and process large matrices.

### 3. Accuracy:

- Ensuring the model accurately reflects user preferences and recommends books that users will genuinely enjoy.
- Challenge: Overcoming biases and ensuring diversity in recommendations.

```
U1  0  5  3  0  0  0  0  4  1  3
U2  0  5  4  0  0  0  0  0  0  0
U3  1  0  0  0  0  3  0  0  0  0
U4  1  0  0  5  5  4  0  0  2  0
U5  0  0  0  5  0  2  0  0  0  0
U6  5  3  5  4  0  5  0  0  0  0
U7  0  0  4  0  0  0  0  4  0  5
U8  3  0  0  4  3  3  3  0  0  0
U9  4  1  0  0  0  0  3  1  4  0
U10 0  3  3  0  3  0  0  0  0  0
```

# Introduction to Singular Value Decomposition (SVD)

## SVD Explained Simply

- What is SVD?
  - A mathematical technique that breaks down a matrix  $A$  into three simpler components:  $U$ ,  $\Sigma$ , and  $V^T$ .

## Components of SVD:

1.  $U$  - User Features Matrix:
  - Captures hidden preferences of users.
2.  $\Sigma$  - Singular Values Matrix:
  - Indicates the importance of each hidden feature.
3.  $V^T$  - Book Features Matrix:
  - Describes characteristics of books.

# Introduction to Singular Value Decomposition (SVD)

**Formula:**

- $A = U\Sigma V^T$ 
  - Decomposes the user-book ratings matrix  $A$ .

**Why Use SVD?**

**Latent Features**

- **Insight:** Uncovers underlying patterns in user ratings.
- **Predictions:** Improves accuracy of book recommendations.

# Latent Features Explained

## What Are Latent Features?

- **Definition:** Hidden characteristics inferred from observable data patterns.

## Role in Data Analysis:

- **Capture Essentials:** Summarize complex data into core elements.
- **Improve Predictions:** Enhance accuracy in systems like recommendation engines.

## Example:

- **Book Ratings:** Latent features could represent genres or themes that aren't explicitly mentioned but influence user preferences.

## SVD Connection:

- **Decomposition:** SVD identifies these features through the matrices:
  - $U$ : User affinities to features.
  - $\Sigma$ : Importance of each feature.
  - $V^T$ : Book alignment with features.

# Applying SVD to a 5x5 Rating Matrix

## Original Rating Matrix $A$

- Represented as:

$$A = \begin{bmatrix} 5 & 3 & 0 & 0 & 0 \\ 4 & 0 & 0 & 3 & 4 \\ 0 & 0 & 0 & 4 & 4 \\ 0 & 3 & 4 & 0 & 0 \\ 0 & 4 & 4 & 0 & 0 \end{bmatrix}$$

## SVD Decomposition

- Decomposed into three matrices  $U$ ,  $\Sigma$ , and  $V^T$ :

$$A = U\Sigma V^T$$

# Applying SVD to a 5x5 Rating Matrix

$$A = U\Sigma V^T$$

Components:

1.  $U$  - User Features Matrix:

$$U = \begin{bmatrix} -0.48 & -0.18 & 0.73 & -0.41 & -0.19 \\ -0.65 & 0.38 & 0.07 & 0.61 & 0.24 \\ -0.43 & 0.39 & -0.56 & -0.56 & -0.21 \\ -0.26 & -0.54 & -0.28 & 0.35 & -0.66 \\ -0.31 & -0.62 & -0.27 & -0.16 & 0.65 \end{bmatrix}$$

2.  $\Sigma$  - Singular Values (Diagonal Matrix):

$$\Sigma = \begin{bmatrix} 8.61 & 0 & 0 & 0 & 0 \\ 0 & 7.74 & 0 & 0 & 0 \\ 0 & 0 & 5.31 & 0 & 0 \\ 0 & 0 & 0 & 1.27 & 0 \\ 0 & 0 & 0 & 0 & 0.18 \end{bmatrix}$$

3.  $V^T$  - Book Features Matrix:

$$V^T = \begin{bmatrix} -0.58 & -0.40 & -0.27 & -0.43 & -0.50 \\ 0.09 & -0.60 & -0.60 & 0.35 & 0.40 \\ 0.74 & 0.06 & -0.41 & -0.38 & -0.36 \\ 0.29 & -0.65 & 0.60 & -0.33 & 0.15 \\ -0.15 & 0.23 & -0.21 & -0.67 & 0.66 \end{bmatrix}$$