NumPy for Data Science Cheat Sheet

Anis Koubaa

October 2024

Companion Notebook

For more detailed examples, visit the Google Colab notebook at: Jupyter Notebook for NumPy Data Analytics.

1 Loading Data

1.1 Reading CSV Files

Data Importing Concept:

- Introduction to loading data from external sources
- Importance of CSV files in data analytics

NumPy Implementation:

- Using NumPy to read CSV files: `data = np.genfromtxt('data.csv', delimiter=',')`
- Handling missing values and setting data types

2 Numpy Array Dimensions and Data Types

2.1 Exploring Array Attributes

Concepts of Array Dimensions and Data Types:

- Understanding the structure of arrays
- The role of data types in numerical computations

NumPy Implementation:

- Examining array dimensions: `print(data.shape)`
- Checking data types: `print(data.dtype)`

3 NumPy Indexing and Slicing

3.1 Accessing Data Elements

Indexing and Slicing Techniques:

- Basics of indexing and slicing in Python arrays
- Advanced techniques for efficient data manipulation

NumPy Implementation:

- Basic indexing: `element = data[0, 0]`
- Slicing arrays: `subarray = data[1:5]`

4 Scalars and NumPy Basics

4.1 Understanding Scalars

Linear Algebra Concept:

- Definition of scalars (single numerical values)
- Operations with scalars

NumPy Implementation:

- Creating scalar variables: `scalar = np.array(5)`
- Performing arithmetic operations with NumPy scalars: `scalar + 2`, `scalar * 3`

5 Vectors

5.1 Introduction to Vectors

Linear Algebra Concept:

- Definition of vectors (ordered lists of numbers)
- Geometric interpretation

NumPy Implementation:

- Representing vectors using 1D arrays: `vector = np.array([1, 2, 3])`
- Initializing vectors: `zeros = np.zeros(3)`, `ones = np.ones(3)`

5.2 Vector Operations

Operations in Linear Algebra:

- Addition and subtraction of vectors
- Scalar multiplication

NumPy Implementation:

- Performing vector addition and subtraction: `result = vector + vector`, `result = vector - vector`
- Scalar multiplication: `scaled = vector * 2`

5.3 Dot Product, Vector Norms, and Cosine Similarity

Linear Algebra Concept:

- Dot product and its properties
- Calculating vector norms (magnitude)
- Understanding cosine similarity between vectors

NumPy Implementation:

- Computing dot product: `dot_product = np.dot(vector, vector)`
- Calculating norms: `norm = np.linalg.norm(vector)`
- Computing cosine similarity: `cosine_similarity = np.dot(vector1, vector2) / (np.linalg.norm(vector1) * np.linalg.norm(vector2))`

6 Matrices

6.1 Understanding Matrices

Linear Algebra Concept:

- Definition of matrices (2D arrays of numbers)
- Applications in transformations

NumPy Implementation:

- Creating matrices with 2D arrays: `matrix = np.array([[1, 2], [3, 4]])`
- Exploring matrix attributes: `print(matrix.shape)`

6.2 Matrix Operations

Operations in Linear Algebra:

- Matrix addition and subtraction
- Scalar multiplication
- Matrix multiplication

NumPy Implementation:

- Performing matrix operations: `result = matrix + matrix`, `result = matrix - matrix`
- Matrix multiplication: `product = np.matmul(matrix, matrix)`

6.3 Special Matrices

Linear Algebra Concept:

- Identity matrix and its properties
- Diagonal matrices

NumPy Implementation:

- Creating identity matrices: `identity = np.eye(3)`
- Extracting diagonals: `diagonal = np.diag(matrix)`

7 Systems of Linear Equations

7.1 Solving Linear Systems

Linear Algebra Concept:

- Representing systems as $Ax = b$
- Methods for solving

NumPy Implementation:

- Solving equations: `x = np.linalg.solve(A, b)`

7.2 Inverse Matrices

Linear Algebra Concept:

- Definition and properties of inverse matrices

NumPy Implementation:

- Calculating inverses: `inv_A = np.linalg.inv(A)`

8 Determinants and Rank

8.1 Determinants

Linear Algebra Concept:

- Understanding determinants and their significance

NumPy Implementation:

- Computing determinants: `det_A = np.linalg.det(A)`

8.2 Rank of a Matrix

Linear Algebra Concept:

- Definition and importance of rank

NumPy Implementation:

- Determining rank: `rank_A = np.linalg.matrix_rank(A)`

9 Eigenvalues and Eigenvectors

9.1 Fundamentals

Linear Algebra Concept:

- What are eigenvalues and eigenvectors?
- Their role in transformations

NumPy Implementation:

- Calculating eigenvalues and eigenvectors: `eigenvalues, eigenvectors = np.linalg.eig(A)`

9.2 Diagonalization

Linear Algebra Concept:

- Diagonalizing a matrix using eigenvalues

NumPy Implementation:

- Performing matrix diagonalization: `diag_matrix = np.diag(eigenvalues)`

10 Singular Value Decomposition (SVD)

10.1 Understanding SVD

Linear Algebra Concept:

- Breaking down a matrix into singular vectors and singular values

NumPy Implementation:

- Performing SVD: `u, s, vh = np.linalg.svd(matrix)`

10.2 Applications of SVD

Concepts:

- Dimensionality reduction
- Noise reduction in data

Practical Application:

- Applying SVD in data science tasks

2 Dataset Overview

This section provides an overview of a dataset used for heart disease studies. We will demonstrate how to load this dataset using both NumPy and Pandas, which are powerful tools for data analysis in Python.

Sample Data Description: Here is a sample of the dataset.

Age	Sex	CP	Trestbps	Chol	FBS	Restecg	Thalach	Exang	Oldpeak	Slope	CA	Thal	Target
63	1	1	145	233	1	2	150	0	2.3	3	0	6	0
67	1	4	160	286	0	2	108	1	1.5	2	3	3	2
67	1	4	120	229	0	2	129	1	2.6	2	2	7	1
37	1	3	130	250	0	0	187	0	3.5	3	0	3	0
41	0	2	130	204	0	2	172	0	1.4	1	0	3	0

2.1 Loading Data with NumPy

NumPy's `genfromtxt` function is perfect for loading data when you don't need to manipulate or preprocess data before analysis.

```
1 import numpy as np
2
3 # Load the heart disease dataset directly into a NumPy array
4 data_np = np.genfromtxt('heart_disease_data.csv', delimiter=',', skip_header=1)
```

2.2 Loading Data with Pandas and Converting to NumPy

Using Pandas to load data provides flexibility for preprocessing. Once data manipulation is complete, you can easily convert the DataFrame to a NumPy array.

```
1 import pandas as pd
2
3 # Load data into a Pandas DataFrame
4 df = pd.read_csv('heart_disease_data.csv')
5 # Convert the DataFrame to a NumPy array
6 data_pd_to_np = df.to_numpy()
```

Here is a sample output, showing how the data looks once loaded:

```
1 array([[67.,  1.,  4., ...,  3.,  3.,  2.],
2        [67.,  1.,  4., ...,  2.,  7.,  1.],
3        [37.,  1.,  3., ...,  0.,  3.,  0.],
4        ...,
5        [57.,  1.,  4., ...,  1.,  7.,  3.],
6        [57.,  0.,  2., ...,  1.,  3.,  1.],
7        [38.,  1.,  3., ..., nan,  3.,  0.]])
```

3 Numpy Array Dimensions and Data Types

After loading the data, it's useful to check the array's shape, data type, and dimensionality. This can provide insight into the structure of the dataset that may influence further data processing steps.

```python
# Print the shape of the data
print('Data Shape:', data_np.shape)

# Print the data type of the array
print('Data Type:', data_np.dtype)

# Print the number of dimensions of the array
print('Number of Dimensions:', data_np.ndim)
```

This will produce the following output, helping to understand the dimensions and type of data stored in the array:

```
Data Shape: (303, 14)  # Assuming there are 303 records and 14 features
Data Type: float64
Number of Dimensions: 2
```

4 NumPy Indexing and Slicing

Indexing and slicing are fundamental for data manipulation in NumPy arrays, allowing for selecting and operating on subsets of the dataset efficiently.

4.1 Basic Indexing

You can access individual elements or ranges of elements in a NumPy array using basic indexing.

```python
# Access the first element (first row)
first_row = data_np[0]

# Access a specific element (element at first row and first column)
first_element = data_np[0, 0]

print("First row:", first_row)
print("First element:", first_element)
```

4.2 Slicing

Slicing allows you to select a subset of your array. This is particularly useful for extracting specific rows or columns.

```python
# Slice the first five rows and the first four columns
subset = data_np[:5, :4]

print("First five rows and four columns:\n", subset)
```

4.3 Boolean Indexing

Boolean indexing is a powerful feature that allows you to select elements based on conditions.

```python
# Find all rows where the target variable (last column) is 1
positive_cases = data_np[data_np[:, -1] == 1]

print("Cases with positive heart disease diagnosis:\n", positive_cases)
```

4.4 Fancy Indexing

Fancy indexing involves passing arrays of indices to access multiple array elements at once.

```
# Access specific rows and columns using lists of indices
rows = [0, 2, 4]  # Select the first, third, and fifth rows
cols = [0, 1, 2, 3]  # Select the first four columns
selected_data = data_np[rows][:, cols]

print("Selected data:\n", selected_data)
```

5 Scalars and NumPy Basics

5.1 Understanding Scalars

Linear Algebra Concept in Data Science: In data science and AI, a scalar is a single value used to adjust or scale features within a dataset. Scalars are fundamental in operations such as feature scaling, which is essential for many machine learning algorithms to perform optimally.

Operations with Scalars:

- **Addition/Subtraction**: Adjusting a baseline value, like correcting sensor offsets in data collection.
- **Multiplication/Division**: Scaling features to a normalized range, such as [0,1] or [-1,1], which is critical for algorithms like gradient descent to converge more efficiently.

NumPy Implementation:

```
import numpy as np

# Creating a scalar value in NumPy
scalar_value = np.array(42)

# Displaying the scalar value and its data type
print("Scalar Value:", scalar_value)
print("Type of Scalar Value:", type(scalar_value))

# For comparison, display the NumPy data type of the scalar
print("NumPy Data Type of Scalar Value:", scalar_value.dtype)

# Assume data_np is a loaded NumPy array from the heart disease dataset
# For instance, adjusting the 'Trestbps' column (index 3) by subtracting the mean

# Calculate the mean of the Trestbps column
mean_trestbps = np.mean(data_np[:, 3])

# Subtract the mean from the Trestbps column to center the data around zero
normalized_trestbps = data_np[:, 3] - mean_trestbps

print("Normalized Trestbps:", normalized_trestbps)
```

This example demonstrates normalizing the resting blood pressure measurements ('Trestbps') by centering them around zero, a common preprocessing step to reduce model bias due to scale differences in features.

6 Vectors

6.1 Introduction to Vectors

Linear Algebra Concept: Vectors provide a way to store and manipulate data across multiple dimensions, essential for numerous algorithms in machine learning and artificial intelligence.

NumPy Implementation: In NumPy, vectors are represented as 1D arrays, which can be used for various mathematical operations.

- Creating vectors using 1D arrays: `vector = np.array([1, 2, 3])`
- Initializing vectors with zeros or ones:
  - `zeros = np.zeros(3)`
  - `ones = np.ones(3)`

6.1.1 Vector Equations

Consider a vector $\vec{v} = [1, 2]$. The compact form of a vector equation might be $\vec{v} = x\vec{u} + y\vec{w}$, where $\vec{u}$ and $\vec{w}$ are unit vectors along the x and y axes, respectively.

Expanded Form:

$$\vec{v} = 1\vec{u} + 2\vec{w}$$

6.1.2 Visual Representation of Vectors in 2D

This figure visually demonstrates how the vector $\vec{v}$ is composed from its components along the x and y axes, illustrating the geometric interpretation of vector addition. The unit vectors $\vec{u}$ and $\vec{w}$ represent the standard basis vectors for the 2D coordinate system, where $\vec{u}$ is aligned with the x-axis and $\vec{w}$ with the y-axis.

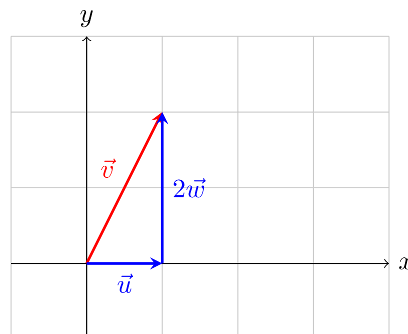

Figure 1: Illustration of vector $\vec{v}$ as the sum of scaled unit vectors $\vec{u}$ along the x-axis and $\vec{w}$ along the y-axis.

6.2 Vector Operations

Operations in Linear Algebra: Vectors can be manipulated through various operations, such as additional and subtraction.

NumPy Implementation: NumPy makes it straightforward to perform these vector operations efficiently, allowing data scientists to manage and manipulate large datasets or model parameters quickly.

- **Performing Vector Addition and Subtraction**: Adding or subtracting vectors element-wise.
- **Scalar Multiplication**: Scaling a vector by multiplying with a scalar value.

Here is how these operations can be implemented in NumPy:

```
import numpy as np

# Creating two vectors
vector1 = np.array([1, 2, 3])
vector2 = np.array([4, 5, 6])

# Vector addition
result_add = vector1 + vector2

# Vector subtraction
result_sub = vector1 - vector2

# Scalar multiplication
scaled = vector1 * 2

print("Vector Addition Result:", result_add)
print("Vector Subtraction Result:", result_sub)
print("Scalar Multiplication Result:", scaled)
```

6.3 Dot Product and Vector Norms

Linear Algebra Concepts: Key operations in vector algebra are utilized in computational geometry, data science, and machine learning, especially for analyzing data in high-dimensional spaces.

- **Dot Product** ($\vec{a} \cdot \vec{b}$): Given two vectors $\vec{a}$ and $\vec{b}$, the dot product is calculated as:

  $$\vec{a} \cdot \vec{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

  This scalar measures the magnitude of $\vec{a}$ in the direction of $\vec{b}$ and is foundational in determining the angle between vectors.

- **Vector Norms** ($||\vec{a}||$): The norm of a vector $\vec{a}$ in Euclidean space is defined as:

  $$||\vec{a}|| = \sqrt{\vec{a} \cdot \vec{a}} = \sqrt{a_1^2 + a_2^2 + \cdots + a_n^2}$$

  Normalizing a vector involves dividing the vector by its norm, resulting in a unit vector. This process is crucial for many machine learning algorithms to ensure unbiased comparisons by distance or angle.

Cosine Similarity: Cosine similarity measures the cosine of the angle between two vectors, providing a scale- and length-independent measure of similarity:

$$\text{Cosine Similarity}(\vec{a}, \vec{b}) = \frac{\vec{a} \cdot \vec{b}}{||\vec{a}|| \times ||\vec{b}||} = \frac{\sum_{i=1}^{n} a_i b_i}{\sqrt{\sum_{i=1}^{n} a_i^2} \times \sqrt{\sum_{i=1}^{n} b_i^2}}$$

This metric ranges from -1 to 1, where 1 indicates vectors in the same direction, 0 indicates orthogonality, and -1 indicates vectors in opposite directions.

NumPy Implementation and Practical Example with Word Embeddings: Word embeddings represent words in multi-dimensional space. By applying dot products and norms, we can objectively measure how similar or different the words are.

```
import numpy as np

# Vectors representing word embeddings for "apple", "banana", and "vehicle"
apple = np.array([1, 2])
banana = np.array([1, 2.1])
vehicle = np.array([8, 3])

# Computing dot products
dot_product_apple_banana = np.dot(apple, banana)
dot_product_apple_vehicle = np.dot(apple, vehicle)

# Calculating norms
norm_apple = np.linalg.norm(apple)
norm_banana = np.linalg.norm(banana)
norm_vehicle = np.linalg.norm(vehicle)

# Calculating cosine similarity
cosine_sim_apple_banana = dot_product_apple_banana / (norm_apple * norm_banana)
cosine_sim_apple_vehicle = dot_product_apple_vehicle / (norm_apple * norm_vehicle)

print("Cosine Similarity between Apple and Banana:", cosine_sim_apple_banana)
print("Cosine Similarity between Apple and Vehicle:", cosine_sim_apple_vehicle)
```

Cosine Similarity: This measure computes the cosine of the angle between two vectors projected in a multi-dimensional space. It is particularly useful in natural language processing to determine how similar two words are, irrespective of their magnitude, by normalizing the vectors to unit length.

Discussion: - The cosine similarity between "apple" and "banana" is higher, indicating a closer relationship due to their similar meanings compared to "apple" and "vehicle." - This example highlights why normalizing vectors is critical for meaningful comparisons in cosine similarity calculations, as it ensures comparisons are based on directions alone, not magnitudes.

This diagram visually demonstrates the positions of "apple," "banana," and "vehicle" in a 2D vector space, reflecting their semantic relationships based on their cosine similarities.

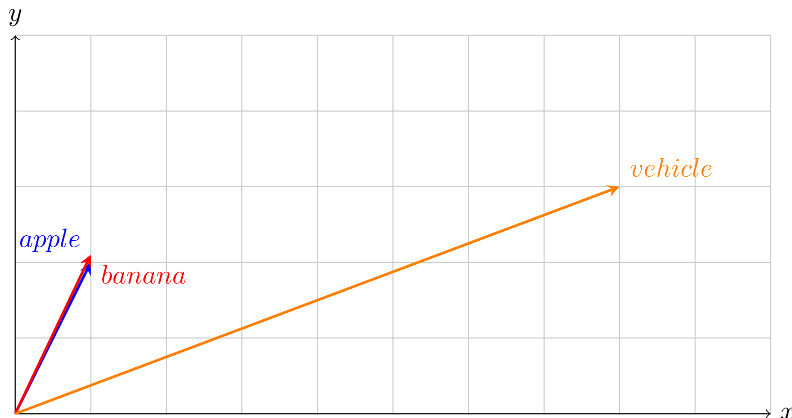

Figure 2: Visualization of word vectors in a 2D space, showing relative distances and directions between 'apple,' 'banana,' and 'vehicle.' These distances and angles illustrate the semantic similarities and differences among the words.

7 Matrices

7.1 Understanding Matrices

Linear Algebra Concept: Matrices are fundamental in linear algebra, representing systems of equations, transformations in graphics, and data structures in machine learning. They facilitate operations across multiple data points simultaneously, making them invaluable in complex computations.

NumPy Implementation: Matrices in NumPy are implemented as 2D arrays, providing a powerful tool for numerical computing with features for performing a wide range of mathematical operations efficiently.

- Creating matrices: `matrix = np.array([[1, 2], [3, 4]])`
- Accessing matrix attributes (e.g., shape, size): `print(matrix.shape)`, `print(matrix.size)`

7.1.1 Matrix Operations

Sample Data Description: Here is a sample of the dataset used to demonstrate matrix operations.

Age	Sex	CP	Trestbps	Chol	FBS	Restecg	Thalach	Exang	Oldpeak	Slope	CA	Thal	Target
63	1	1	145	233	1	2	150	0	2.3	3	0	6	0
67	1	4	160	286	0	2	108	1	1.5	2	3	3	2
67	1	4	120	229	0	2	129	1	2.6	2	2	7	1
37	1	3	130	250	0	0	187	0	3.5	3	0	3	0
41	0	2	130	204	0	2	172	0	1.4	1	0	3	0

NumPy Implementation:

- **Loading Data into a Matrix:** To use this dataset in NumPy, you would typically load it from a CSV file using:

```python
import numpy as np
data = np.genfromtxt('heart_disease_data.csv', delimiter=',', skip_header=1)

```

- **Matrix Multiplication:** Demonstrates matrix multiplication which is crucial for data transformations:

```python
# Assuming 'matrix' is defined as np.array([[1, 2], [3, 4]])
# Multiply by a vector
vector = np.array([1, 1])
product = np.dot(matrix, vector)  # Output will be an array([3, 7])

```

7.1.2 Practical Application: Data Transformation

Matrix operations are core to many algorithms in data science. For example, transforming data features for standardization:

```python
# Standardizing the 'Age' column (index 0 in the dataset)
age_mean = np.mean(data[:,0])
age_std = np.std(data[:,0])
standardized_age = (data[:,0] - age_mean) / age_std
```

This section illustrates the depth of matrix manipulation capabilities in NumPy, using a practical example from the dataset that includes basic loading, manipulation, and application in data science tasks like feature scaling.

7.2 Matrix Operations

Operations in Linear Algebra: Matrix operations are crucial for many applications in engineering, physics, economics, and computer science, allowing for the manipulation and transformation of data stored in matrices.

- **Matrix Addition and Subtraction:**
  Compact Form:
  $$C = A \pm B$$
  Expanded Form:
  $$C = \begin{bmatrix} a_{11} \pm b_{11} & a_{12} \pm b_{12} \\ a_{21} \pm b_{21} & a_{22} \pm b_{22} \end{bmatrix}$$
- **Scalar Multiplication:**
  Compact Form:
  $$C = kA$$
  Expanded Form:
  $$C = \begin{bmatrix} k \cdot a_{11} & k \cdot a_{12} \\ k \cdot a_{21} & k \cdot a_{22} \end{bmatrix}$$

- **Matrix Multiplication:**
  Compact Form:
  $$C = AB$$
  Expanded Form:
  $$C = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}$$
- **Dot Product between a Matrix and a Vector:** This operation involves multiplying a matrix by a vector. The result is a new vector where each component is a linear combination of the columns of the matrix, weighted by the corresponding components of the vector.
  Compact Form:
  $$\vec{u} = A\vec{v}$$
  Expanded Form: Assume a 2x2 matrix A and a 2-dimensional vector $\vec{v}$:
  $$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, \quad \vec{v} = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$$
  $$\vec{u} = \begin{bmatrix} a_{11}v_1 + a_{12}v_2 \\ a_{21}v_1 + a_{22}v_2 \end{bmatrix}$$
  NumPy Implementation: NumPy provides intuitive and powerful tools for performing these operations efficiently, essential for numerical computing and data analysis tasks.
  - **Performing Matrix Addition and Subtraction:**
```
# Assuming matrices A and B
result_add = A + B
result_sub = A - B

```
  - **Scalar Multiplication:**
```
# Scalar multiplication by a constant c
result_scalar_mult = A * c

```
  - **Matrix Multiplication:**
```
# Matrix multiplication of A and B
result_mat_mult = np.dot(A, B)

```
  - **Cross Product:**
```
# Cross product of vectors a and b from matrices A and B
result_cross_prod = np.cross(a, b)

```
  - **Dot Product:**
```
# Dot product of matrix A and vector v
result_dot_prod = np.dot(A, v)

```

7.3 Special Matrices

Linear Algebra Concept: Special matrices are utilized extensively in mathematical computations due to their unique properties.

- **Identity Matrix:** An identity matrix is a square matrix with ones on the main diagonal and zeros elsewhere. It functions as the multiplicative identity in matrix multiplication, such that any matrix multiplied by an identity matrix does not change.
- **Diagonal Matrix:** A diagonal matrix is a type of matrix in which all off-diagonal entries are zero. The non-zero entries can be found only on the diagonal running from the upper left to the lower right.

NumPy Implementation: NumPy provides efficient functions to handle these types of matrices, useful in various numerical computations.

- **Creating Identity Matrices:**

```
identity = np.eye(3)
```

  This function creates a 3x3 identity matrix:

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

- **Extracting Diagonals:**

```
diagonal = np.diag(matrix)
```

  If `matrix` is:

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix},$$

  then `np.diag(matrix)` will return:

$$\begin{bmatrix} 1 & 5 & 9 \end{bmatrix}$$

Example of Matrix Multiplication with an Identity Matrix: Consider a 3x3 matrix A:

$$A = \begin{bmatrix} 2 & 3 & 4 \\ 5 & 6 & 7 \\ 8 & 9 & 10 \end{bmatrix}$$

Multiplying A by the identity matrix I results in:

$$A \cdot I = \begin{bmatrix} 2 & 3 & 4 \\ 5 & 6 & 7 \\ 8 & 9 & 10 \end{bmatrix}$$

NumPy Implementation and Examples: NumPy provides intuitive functions to create and manipulate these matrices. Below are some Python code examples that illustrate their creation and use:

```python
import numpy as np

# Creating a 3x3 identity matrix
identity = np.eye(3)
print("Identity Matrix:\\n", identity)

# Creating a matrix to demonstrate diagonal extraction
matrix = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
diagonal = np.diag(matrix)
print("Diagonal of the Matrix:\\n", diagonal)

# Example of Matrix Multiplication with an Identity Matrix
A = np.array([[2, 3, 4], [5, 6, 7], [8, 9, 10]])
result = np.dot(A, identity)
print("Result of Multiplying A by the Identity Matrix:\\n", result)
```

8 Systems of Linear Equations

8.1 Solving Linear Systems

Linear Algebra Concept: Systems of linear equations can be compactly represented as matrix equations of the form $Ax = b$, where A is a coefficient matrix, x is a vector of variables, and b is a vector of constants.

Illustrative Numerical Example and NumPy Implementation: Consider the system of linear equations given by:

$$3x_1 + x_2 = 9$$

$$x_1 + 2x_2 = 8$$

This system can be represented in matrix form as $Ax = b$, where

$$A = \begin{bmatrix} 3 & 1 \\ 1 & 2 \end{bmatrix}, \quad b = \begin{bmatrix} 9 \\ 8 \end{bmatrix}$$

The matrix A contains the coefficients of the variables x_1 and x_2, and vector b contains the constants from the right-hand side of the equations.

To find the solution vector x, which contains the values of x_1 and x_2, we solve the matrix equation:

$$x = A^{-1}b$$

Using NumPy to perform this computation:

```python
import numpy as np

# Define the matrix A and vector b
A = np.array([[3, 1], [1, 2]])
b = np.array([9, 8])

# Use NumPy to solve for x
x = np.linalg.solve(A, b)

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

Executing the above Python code yields:

$$x = \begin{bmatrix} 2 \\ 3 \end{bmatrix}$$

This means that $x_1 = 2$ and $x_2 = 3$ are the solutions to the system of equations, effectively satisfying both equations. This illustrative example not only demonstrates how to set up and solve a system of linear equations in matrix form but also confirms the solution using a direct computational approach with NumPy.

8.2 Inverse Matrices

Linear Algebra Concept: The inverse of a matrix A, denoted as A^{-1}, is a matrix that, when multiplied by A, yields the identity matrix I. For a matrix to have an inverse, it must be a square matrix (i.e., the same number of rows and columns) and non-singular, meaning its determinant is not zero ($\det(A) \neq 0$). This non-zero determinant ensures that the matrix has linearly independent columns, which is a prerequisite for invertibility.

NumPy Implementation: NumPy provides the 'linalg.inv' function to compute the inverse of an invertible matrix.

```
import numpy as np

# Define the matrix A
A = np.array([[3, 1], [1, 2]])

# Calculate the inverse of A
inv_A = np.linalg.inv(A)
print("Inverse of A:\\n", inv_A)
```

Illustrative Numerical Example: For the matrix A given above, its inverse A^{-1} can be computed:

$$A^{-1} = \begin{bmatrix} 0.4 & -0.2 \\ -0.2 & 0.6 \end{bmatrix}$$

The product $A \cdot A^{-1}$ confirms the identity matrix:

$$A \cdot A^{-1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

This computation illustrates that multiplying A by A^{-1} yields the identity matrix, verifying the properties of inverse matrices.

8.3 Pseudo Inverse

Linear Algebra Concept: The pseudo-inverse, or Moore-Penrose inverse, of a matrix is particularly useful for matrices that are not square or are singular. It is denoted as A^+ and is used to compute solutions to systems of linear equations that may not have a unique solution. This is often used in least squares data fitting and calculating optimal solutions to overdetermined systems.

NumPy Implementation: NumPy provides the 'linalg.pinv' function to compute the pseudo-inverse of any matrix, even if it is not invertible.

```python
import numpy as np

# Define a matrix B, possibly non-square or singular
B = np.array([[1, 2], [3, 4], [5, 6]])

# Calculate the pseudo-inverse of B
pseudo_inv_B = np.linalg.pinv(B)
print("Pseudo-inverse of B:\\n", pseudo_inv_B)
```

Illustrative Numerical Example: The pseudo-inverse B^+ for the matrix B is:

$$B^+ = \begin{bmatrix} -1.944 & -0.444 & 1.056 \\ 1.611 & 0.389 & -0.833 \end{bmatrix}$$

This matrix, when used to multiply B, approximates an identity matrix, reflecting the least squares solution to an overdetermined system. This concept is crucial in applications like signal processing and regression analysis where exact solutions are not feasible.

9 Determinants and Rank

9.1 Determinants

Linear Algebra Concept: The determinant is a scalar value that can be calculated from the elements of a square matrix. It provides insights into the matrix's properties, indicating whether the matrix is invertible (a determinant of zero means the matrix is not invertible) and the volume factor of the linear transformation represented by the matrix.

General Equation for Determinants: For a matrix

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

the determinant can be calculated using the Leibniz formula, which is a sum over permutations of the matrix indices:

$$\det(A) = \sum_{\sigma \in S_n} \operatorname{sgn}(\sigma) \prod_{i=1}^{n} a_{i,\sigma(i)}$$

where σ ranges over all permutations of n elements, and $\operatorname{sgn}(\sigma)$ is the sign of the permutation ($+1$ for even permutations and -1 for odd permutations).

NumPy Implementation:

```python
import numpy as np

# Define an n x n matrix
A = np.array([[a11, a12, ..., a1n], [a21, a22, ..., a2n], ..., [an1, an2, ..., ann
    ]])
det_A = np.linalg.det(A)
print("Determinant of A:", det_A)
```

9.2 Rank of a Matrix

Linear Algebra Concept: The rank of a matrix is defined as the maximum number of linearly independent rows or columns, which is also the dimension of the row space or column space of the matrix.

General Equation for Rank: The rank of an $m \times n$ matrix A can be determined by reducing the matrix to its Row Echelon Form (REF) and counting the non-zero rows. Mathematically, this process involves elementary row operations that do not change the row space of the matrix.

NumPy Implementation:

```python
import numpy as np

# Define an m x n matrix A
A = np.array([[4, 5, 6], [7, 8, 9], [1, 1, 1]])
rank_A = np.linalg.matrix_rank(A)
print("Rank of A:", rank_A)
```

9.3 Trace of a Matrix

Linear Algebra Concept: The trace of a square matrix is the sum of the elements on its main diagonal. The trace operation is important because it is invariant under change of basis and gives the sum of the eigenvalues of the matrix.

General Equation for Trace: For a square matrix

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

the trace of A, denoted as $\text{tr}(A)$, is calculated as:

$$\text{tr}(A) = a_{11} + a_{22} + \cdots + a_{nn}$$

NumPy Implementation:

```python
import numpy as np

# Define a square matrix A
A = np.array([[4, 5, 6], [7, 8, 9], [10, 11, 12]])
trace_A = np.trace(A)
print("Trace of A:", trace_A)
```

10 Eigenvalues and Eigenvectors

10.1 Fundamentals

Linear Algebra Concept: Eigenvalues (λ) and eigenvectors ($\mathbf{v}$) of a square matrix A are fundamental in understanding the behavior of linear transformations represented by A. The eigenvector $\mathbf{v}$ is a non-zero vector that only scales (by λ) when the matrix A is applied to it, thus satisfying:

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

This relationship can be explored by rearranging the terms:

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

where I is the identity matrix of the same dimension as A. This equation implies that for non-trivial solutions, the matrix $(A - \lambda I)$ must be singular, which leads to the characteristic equation:

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

Solving this polynomial in λ gives the eigenvalues, and substituting each λ back into the equation provides the corresponding eigenvectors.

NumPy Implementation:

```
import numpy as np

# Define a matrix A
A = np.array([[4, 2], [1, 3]])

# Calculate eigenvalues and eigenvectors
eigenvalues, eigenvectors = np.linalg.eig(A)
print("Eigenvalues:", eigenvalues)
print("Eigenvectors:\\n", eigenvectors)
```

Numerical Example: Consider the matrix A:

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

Finding Eigenvalues: To find the eigenvalues λ, solve the characteristic equation:

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

First, set up $A - \lambda I$:

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

The determinant of this matrix is:

$$\det(A - \lambda I) = (4 - \lambda)(3 - \lambda) - 2 \cdot 1 = \lambda^2 - 7\lambda + 10$$

Solve the quadratic equation $\lambda^2 - 7\lambda + 10 = 0$ using the quadratic formula:

$$\lambda = \frac{7 \pm \sqrt{49 - 40}}{2} = \frac{7 \pm 3}{2}$$

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

Finding Eigenvectors: For each eigenvalue, solve $(A - \lambda I)\mathbf{v} = 0$ to find the corresponding eigenvectors.

Eigenvector for $\lambda_1 = 5$:

$$A - 5I = \begin{bmatrix} -1 & 2 \\ 1 & -2 \end{bmatrix}$$

Row reduce this matrix:

$$\begin{bmatrix} -1 & 2 \\ 1 & -2 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & -2 \\ 0 & 0 \end{bmatrix}$$

From $x_1 - 2x_2 = 0$, let $x_2 = t$. Then $x_1 = 2t$.

$$\mathbf{v}_1 = t \begin{bmatrix} 2 \\ 1 \end{bmatrix} \quad (\text{where } t \neq 0, \text{ typically } t = 1)$$

Choosing $t = 1$ gives:

$$\mathbf{v}_1 = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

Eigenvector for $\lambda_2 = 2$:

$$A - 2I = \begin{bmatrix} 2 & 2 \\ 1 & 1 \end{bmatrix}$$

Row reduce this matrix:

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

From $x_1 + x_2 = 0$, let $x_2 = t$. Then $x_1 = -t$.

$$\mathbf{v}_2 = t \begin{bmatrix} -1 \\ 1 \end{bmatrix} \quad (\text{where } t \neq 0, \text{ typically } t = 1)$$

Choosing $t = 1$ gives:

$$\mathbf{v}_2 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$$

10.2 Diagonalization

Linear Algebra Concept: Diagonalization is the process of transforming a matrix into a diagonal form that reflects its eigenvalues, which is possible if the matrix has enough linearly independent eigenvectors to form a basis. A matrix A is diagonalizable if it can be expressed as:

$$A = PDP^{-1}$$

where P is the matrix of column eigenvectors of A, and D is the diagonal matrix containing the eigenvalues of A. Each column $\mathbf{v}_i$ of P corresponds to an eigenvalue λ_i in D.

NumPy Implementation:

```
# Assuming eigenvalues and eigenvectors from previous code
P = eigenvectors
D = np.diag(eigenvalues)
P_inv = np.linalg.inv(P)
A_diag = np.dot(np.dot(P, D), P_inv)
print("Diagonalized matrix A:\\n", A_diag)
```

Verification of Diagonalization: With the eigenvalues $\lambda_1 = 5$ and $\lambda_2 = 2$ and the corresponding eigenvectors $\mathbf{v}_1 = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$ and $\mathbf{v}_2 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$ computed previously, we can perform matrix diagonalization and verify it by reconstructing the original matrix A.

Setup the Matrix P and D: Matrix P, containing the eigenvectors as columns:

$$P = \begin{bmatrix} 2 & -1 \\ 1 & 1 \end{bmatrix}$$

Matrix D, a diagonal matrix containing the eigenvalues:

$$D = \begin{bmatrix} 5 & 0 \\ 0 & 2 \end{bmatrix}$$

Calculate P^{-1}: Inverse of P:

$$P^{-1} = \frac{1}{\det(P)} \begin{bmatrix} 1 & 1 \\ -1 & 2 \end{bmatrix} = \frac{1}{3} \begin{bmatrix} 1 & 1 \\ -1 & 2 \end{bmatrix}$$

Reconstruct A by Calculating PDP^{-1}: Multiply the matrices to reconstruct A:

$$A = PDP^{-1} = \begin{bmatrix} 2 & -1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 5 & 0 \\ 0 & 2 \end{bmatrix} \frac{1}{3} \begin{bmatrix} 1 & 1 \\ -1 & 2 \end{bmatrix}$$

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

11 Principal Component Analysis (PCA)

11.1 Understanding PCA

Linear Algebra Concept:

- PCA is a technique for reducing the dimensionality of data while preserving as much variance as possible.
- It involves transforming the data into a new coordinate system where the axes (principal components) are directions of maximum variance.
- This transformation is achieved by finding the eigenvalues and eigenvectors of the covariance matrix of the data.

Steps to Perform PCA:

1. **Center the Data:** Subtract the mean of each feature from the dataset X to obtain a mean-centered matrix X_c.
2. **Compute the Covariance Matrix:**

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

   where X_c is the mean-centered data matrix and n is the number of samples.
3. **Find Eigenvalues and Eigenvectors:** Compute the eigenvalues λ_i and eigenvectors $\mathbf{v}_i$ of the covariance matrix.

4. **Sort and Select Principal Components:** Sort the eigenvalues in descending order and select the top k eigenvectors to form the transformation matrix.

5. **Transform the Data:** Project the data onto the new subspace:

$$X_{\text{PCA}} = X_c \cdot V_k$$

where V_k is the matrix of the top k eigenvectors.

NumPy Implementation:

```
import numpy as np

# Define a mean-centered data matrix X_c (3x2 for simplicity)
X_c = np.array([[1, 2], [3, 4], [5, 6]]) - np.mean(np.array([[1, 2], [3, 4], [5,
    6]]), axis=0)

# Compute the covariance matrix
cov_matrix = np.cov(X_c, rowvar=False)

# Compute eigenvalues and eigenvectors
eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)
print("Covariance Matrix:\\n", cov_matrix)
print("Eigenvalues:", eigenvalues)
print("Eigenvectors:\\n", eigenvectors)

# Select the top k eigenvectors (here, k=1)
k = 1
V_k = eigenvectors[:, :k]

# Transform the data
X_pca = X_c @ V_k
print("Transformed data:\\n", X_pca)
```

11.2 Covariance Matrix and Eigenvalues

Mathematical Explanation:

- The covariance matrix captures the variance and the relationship between different features of the mean-centered data:

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

- The eigenvalues λ_i represent the variance captured by each principal component.
- The corresponding eigenvectors $\mathbf{v}_i$ indicate the direction of each principal component.

Numerical Example: Consider the covariance matrix $\text{Cov}(X_c)$ calculated from the centered data:

$$\text{Cov}(X_c) = \begin{bmatrix} \sigma_{11} & \sigma_{12} \\ \sigma_{21} & \sigma_{22} \end{bmatrix} = \begin{bmatrix} 8.33 & 8.33 \\ 8.33 & 8.33 \end{bmatrix}$$

Calculating the eigenvalues and eigenvectors:

$$\lambda_1 = 16.66, \quad \lambda_2 = 0$$

$$\mathbf{v}_1 = \begin{bmatrix} \frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} \end{bmatrix}, \quad \mathbf{v}_2 = \begin{bmatrix} -\frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} \end{bmatrix}$$

11.3 Applications of PCA

Concepts:

- **Dimensionality Reduction:** PCA projects the data onto the top k principal components to reduce dimensionality while preserving variance.
- **Feature Extraction:** Identifies new axes of maximum variance, allowing for improved data representation and simplification.

Practical Application:

- Apply PCA to reduce data dimensionality, simplify models, and visualize high-dimensional datasets.

Reconstructing Data:

- The approximate reconstruction of the original data using the top k principal components is:

$$X_{\text{reconstructed}} = X_{\text{PCA}} \cdot V_k^T + \mu$$

where μ is the mean vector used for centering.

12 Singular Value Decomposition (SVD)

12.1 Understanding SVD

Linear Algebra Concept:

- Singular Value Decomposition (SVD) is a factorization of a matrix into three components:

$$A = U\Sigma V^T$$

where A is an $m \times n$ matrix.

  - U is an $m \times m$ orthogonal matrix whose columns are the left-singular vectors of A.
  - Σ is an $m \times n$ diagonal matrix with non-negative real numbers on the diagonal known as singular values.
  - V^T is the transpose of an $n \times n$ orthogonal matrix whose columns are the right-singular vectors of A.
- The singular values in Σ are the square roots of the eigenvalues of both A^TA and AA^T.

NumPy Implementation:

```
1 import numpy as np
2
3 # Define a matrix A
4 matrix = np.array([[1, 2], [3, 4], [5, 6]])
5
6 # Perform Singular Value Decomposition
7 u, s, vh = np.linalg.svd(matrix)
8 print("U matrix:\\n", u)
9 print("Singular values:\\n", s)
10 print("V^T matrix:\\n", vh)
```

12.2 Applications of SVD

Concepts:

- **Dimensionality Reduction:** SVD is used to reduce the number of features in data while retaining the most critical information. This is achieved by truncating Σ to keep only the top k largest singular values and corresponding columns of U and rows of V^T:

$$A_k = U_k \Sigma_k V_k^T$$

  where U_k, Σ_k, and V_k^T are truncated versions of U, Σ, and V^T.

- **Noise Reduction:** Smaller singular values often correspond to noise or less significant information. Discarding these values helps in cleaning the data, enhancing pattern recognition and interpretation.

Practical Application:

```
# Assuming top 1 singular value to reduce dimensionality
k = 1
A_k = np.dot(u[:, :k] * s[:k], vh[:k, :])
print("Approximated matrix A_k:\\n", A_k)
```

Given the matrix A:

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

12.3 Performing SVD

Performing the Singular Value Decomposition of A yields the matrices U, Σ, and V^T, where:

Matrix U (Left-singular vectors):

$$U = \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \\ u_{31} & u_{32} & u_{33} \end{bmatrix}$$

Singular values (arranged in Σ):

$$\Sigma = \begin{bmatrix} \sigma_1 & 0 & 0 & 0 \\ 0 & \sigma_2 & 0 & 0 \\ 0 & 0 & \sigma_3 & 0 \end{bmatrix}$$

Matrix V^T (Right-singular vectors):

$$V^T = \begin{bmatrix} v_{11} & v_{12} & v_{13} & v_{14} \\ v_{21} & v_{22} & v_{23} & v_{24} \\ v_{31} & v_{32} & v_{33} & v_{34} \end{bmatrix}$$

12.4 Matrix Reconstruction

Using the SVD components, the original matrix A can be reconstructed as follows:

$$A = U\Sigma V^T$$

Expanding this product, we express A as a sum of outer products scaled by the singular values:

$$A = \sigma_1 \mathbf{u}_1 \mathbf{v}_1^T + \sigma_2 \mathbf{u}_2 \mathbf{v}_2^T + \sigma_3 \mathbf{u}_3 \mathbf{v}_3^T$$

where $\mathbf{u}_i$ and $\mathbf{v}_i$ are columns of U and rows of V^T, respectively.

12.5 Numercial Example

Performing the Singular Value Decomposition of A yields the matrices U, Σ, and V^T, with the following results:

Matrix U (Left-singular vectors):

$$U = \begin{bmatrix} 0.20673589 & 0.88915331 & 0.40824829 \\ 0.51828874 & 0.25438183 & -0.81649658 \\ 0.82984158 & -0.38038964 & 0.40824829 \end{bmatrix}$$

Singular values (arranged in Σ):

$$\Sigma = \begin{bmatrix} 2.54368356e+01 & 0 & 0 & 0 \\ 0 & 1.72261225e+00 & 0 & 0 \\ 0 & 0 & 5.14037515e-16 & 0 \end{bmatrix}$$

Matrix V^T (Right-singular vectors):

$$V^T = \begin{bmatrix} 0.40361757 & 0.46474413 & 0.52587069 & 0.58699725 \\ -0.73286619 & -0.28984978 & 0.15316664 & 0.59618305 \\ 0.44527162 & -0.83143156 & 0.32704826 & 0.05911168 \end{bmatrix}$$

12.6 Matrix Reconstruction

Using the SVD components, the original matrix A can be reconstructed as follows:

$$A = U\Sigma V^T$$

Expanding this product, the reconstruction of A involves summing the outer products of the singular vectors scaled by the corresponding singular values:

$$A = 25.4368356 \cdot \begin{bmatrix} 0.20673589 \\ 0.51828874 \\ 0.82984158 \end{bmatrix} \begin{bmatrix} 0.40361757 & 0.46474413 & 0.52587069 & 0.58699725 \end{bmatrix}^T + 1.72261225 \cdot \begin{bmatrix} 0.88915331 \\ 0.25438183 \\ -0.38038964 \end{bmatrix} \begin{bmatrix} -0.7$$

13 Interpretation of SVD Components

13.1 Matrix U (Left-Singular Vectors)

- **Mathematical Significance:** U contains the left-singular vectors of A, forming an orthogonal matrix. These vectors define orthonormal bases for the row space transformations of A.
- **Practical Interpretation:** In data science, the columns of U represent the principal directions of data variation when projected into a lower-dimensional space, ordered by importance.

13.2 Matrix Σ (Singular Values)

- **Mathematical Significance:** Σ is a diagonal matrix with singular values which are the square roots of the eigenvalues of $A^T A$. These values are sorted in descending order.
- **Practical Interpretation:** Singular values quantify the contribution of each singular vector to the overall data structure. They measure the "strength" or "information content" of each principal direction.

13.3 Matrix V^T (Right-Singular Vectors)

- **Mathematical Significance:** V^T, containing the right-singular vectors, is another orthogonal matrix. It represents transformations within the column space of A.
- **Practical Interpretation:** The rows of V^T (columns of V) are weights for combining the principal components to reconstruct original features, revealing underlying patterns in data.

13.4 Application in Reconstruction

- Combining U, Σ, and V^T reconstructs the original matrix A or its approximation by reducing dimensions, thereby summarizing essential features with minimal data loss, used extensively for noise reduction and data compression.