A Foundational Guide to NumPy and Matplotlib for Machine Learning

Part I: The Bedrock of Scientific Computing - NumPy

This first part of the report establishes the critical role of NumPy as the foundational library for numerical computation in Python, particularly within the context of machine learning. It moves from the core motivation for NumPy's existence to the practical skills required to create, inspect, and manipulate its central data structure, the ndarray.

Section 1: Introduction to NumPy: The "Why" and "What" for Machine Learning

Before diving into the practicalities of machine learning algorithms, it is essential to master the tools that make them possible. In the Python ecosystem, the single most important library for numerical work is NumPy (Numerical Python). It serves as the bedrock upon which the entire scientific Python stack, including key machine learning libraries, is built. This section explains why NumPy is not just a convenience but a necessity for performance-driven data science.

1.1 The Need for Speed and Efficiency

A common question for those new to the field is, "Why not just use standard Python lists?" While Python's built-in lists are versatile general-purpose containers, they are fundamentally ill-suited for the large-scale numerical operations that characterize machine learning.[4] Their flexibility comes at a significant performance cost. Each element in a Python list is a complete

PyObject, which carries not only its value but also substantial overhead, including type information and a reference count.[5] This structure makes lists heterogeneous, meaning they can store different data types (e.g., an integer, a string, and a float) in the same collection.

NumPy's solution to this performance bottleneck is its core data structure: the **N-dimensional array**, or ndarray.[1] The defining feature of an

ndarray is its homogeneity; all elements within the array must be of the same data type.[1] This constraint is the key to unlocking massive performance gains. By sacrificing the data-type flexibility of lists, NumPy can store data in a much more efficient manner, which is crucial for handling the large, uniform datasets typical in machine learning, such as an image represented as a matrix of pixel values or a dataset composed of numerical features.

1.2 The Architectural Advantage: Contiguous Memory and C-Level Performance

The performance advantage of NumPy is not merely incremental; it is architectural. The elements of an ndarray are stored in a single, continuous (contiguous) block of memory.[5] This is in stark contrast to Python lists, which are effectively arrays of pointers, with each pointer directing to a

PyObject that could be located anywhere in memory.[6]

This contiguous memory layout enables several critical optimizations:

1. **Locality of Reference:** When data is stored together, the processor can make more effective use of its cache. Instead of chasing pointers all over memory, the CPU can load a large chunk of the array into its high-speed cache at once, dramatically speeding up access to sequential elements.[6]
2. **Vectorization and SIMD:** Contiguous memory allows NumPy to leverage Single Instruction, Multiple Data (SIMD) instructions available on modern CPUs. This means a single operation (like addition or multiplication) can be applied to multiple array elements simultaneously, rather than one at a time.[7]
3. **Optimized C Loops:** The core operations and mathematical functions in NumPy are not executed in Python's interpreted loops. Instead, they are highly optimized,

pre-compiled loops written in C or Fortran.[5] This completely bypasses the overhead of the Python interpreter for the most computationally intensive parts of a program.

The cumulative effect of these optimizations is profound. Numerical operations on NumPy arrays can be up to 50 times faster than their equivalents performed on Python lists, a performance gap that becomes increasingly critical as dataset sizes grow into the thousands or millions of samples.[4] The choice between a list and an array represents a fundamental trade-off in scientific computing: lists offer flexibility, while NumPy arrays offer the raw performance necessary for machine learning. For numerical data, the performance gain from homogeneity and contiguous storage far outweighs the need for data type flexibility within a single array structure.

1.3 The Foundation of the ML Ecosystem

NumPy's importance extends beyond its own capabilities. It serves as the universal data structure and computational engine for the broader scientific Python ecosystem.[3] Libraries that are indispensable for machine learning, such as

Pandas (for data manipulation), **Matplotlib** (for visualization), and **Scikit-learn** (for ML algorithms), are all built on top of and designed to operate seamlessly with NumPy arrays.[3] Data is passed between these libraries in the form of

ndarrays. Therefore, a deep understanding of NumPy is not just recommended; it is a prerequisite for any serious work in machine learning with Python.

To provide a clear summary, the following table contrasts the key features of Python lists and NumPy arrays.

Feature	Python List	NumPy ndarray	Implication for Machine Learning
Memory Layout	Scattered; array of pointers to objects [6]	Contiguous block of memory [6]	Contiguous memory enables massive speedups via CPU caching and vectorization.

Data Types	Heterogeneous (mixed types allowed)	Homogeneous (all elements same type) [1]	Homogeneity is a prerequisite for high-performance numerical operations on large datasets.
Performance	Slow for numerical operations due to interpreter overhead [5]	Extremely fast; operations executed in compiled C code [5]	Essential for training models on large datasets in a reasonable amount of time.
Functionality	General-purpose container; limited math functions [5]	Vast library of optimized functions for math and linear algebra [4]	Provides the mathematical toolkit (e.g., dot products, matrix operations) needed for ML.
Memory Usage	High overhead per element	Compact; minimal memory overhead per element [5]	More memory-efficient for storing large volumes of numerical data.

Section 2: Array Creation: Your First Steps with NumPy

With the motivation for using NumPy established, the next step is to learn how to create its core object, the ndarray. NumPy provides a rich set of functions for creating arrays, whether from existing data or from scratch.

2.1 Getting Started: Installation and Importing

Before using NumPy, it must be installed in the Python environment. This is typically done using pip, the Python package installer.[2]

Bash

```
pip install numpy
```

Once installed, the library must be imported into a Python script or session. The universal convention, followed by the entire data science community, is to import NumPy with the alias np.[3]

```
Python

import numpy as np
```

2.2 From Python to NumPy: Creating Arrays from Existing Data

The most straightforward way to create a NumPy array is by converting an existing Python sequence, such as a list or a tuple, using the np.array() function.[1] The structure of the list determines the dimensionality of the resulting array.

```
Python

# Create a 1D array from a list
list_1d =
arr_1d = np.array(list_1d)
print(arr_1d)
# Output: [1 2 3 4]

# Create a 2D array (matrix) from a list of lists
list_2d = [, ]
arr_2d = np.array(list_2d)
print(arr_2d)
# Output:
# [[1 2 3]
#  [4 5 6]]
```

A critical parameter in array creation is dtype, which allows for explicit control over the data type of the array's elements.[16] This is vital for managing memory usage and numerical precision in machine learning models. If not specified, NumPy infers the type from the input data. If a list contains both integers and floating-point numbers, NumPy will upcast the entire array to a floating-point type to maintain homogeneity.[3]

Python

```python
# Explicitly create a float array from integers
arr_float = np.array(, dtype=np.float64)
print(arr_float)
print(arr_float.dtype)
# Output:
# [1. 2. 3.]
# float64

# NumPy automatically upcasts to float
mixed_arr = np.array([1, 2.5, 3])
print(mixed_arr.dtype)
# Output: float64
```

2.3 Intrinsic Array Creation: Building from Scratch

Often in machine learning, it is necessary to create arrays without pre-existing data, for instance, to initialize weight matrices in a neural network or to create placeholder arrays.

- np.zeros() and np.ones(): These functions create arrays of a given shape filled entirely with 0s or 1s, respectively. By default, they create floating-point arrays.[1]
- np.full(): Creates an array of a given shape filled with a specified constant value.[3]
- np.eye(): Creates a 2D identity matrix (1s on the diagonal, 0s elsewhere), which is a fundamental tool in linear algebra.[13]
- np.empty(): This function allocates the memory for an array of a given shape but does not initialize the values. Its contents will be whatever data was already in that

memory location. It is slightly faster than np.zeros() but should be used with caution, ensuring that every element is explicitly assigned a value before being used.[1]

Python

```python
# Create a 2x3 array of zeros
zeros_arr = np.zeros((2, 3))
print(zeros_arr)
# Output:
# [[0. 0. 0.]
#  [0. 0. 0.]]

# Create a 3x3 identity matrix
identity_matrix = np.eye(3)
print(identity_matrix)
# Output:
# [[1. 0. 0.]
#  [0. 1. 0.]
#  [0. 0. 1.]]
```

2.4 Creating Numerical Sequences

NumPy provides two primary functions for creating numerical sequences, a common task for generating feature values or coordinates for plotting. The choice between them depends on whether one needs to control the step size or the total number of points.

- np.arange(start, stop, step): This function is analogous to Python's built-in range but returns a NumPy array instead of an iterator. It generates values in a half-open interval [start, stop) with a specified step.[1]
- np.linspace(start, stop, num): This function is often more useful for scientific applications. It generates a specified num of values evenly spaced over a closed interval [start, stop]. The key difference is that its primary control is the number of points, not the step size.[1]

The distinction between these two functions is subtle but important. When working with floating-point numbers, np.arange() can suffer from precision issues, where the stop value might be unexpectedly included or excluded due to rounding errors.[19]

np.linspace() avoids this ambiguity by guaranteeing the exact number of points and the inclusion of the endpoints, making it the more robust and professionally preferred choice for generating non-integer sequences, such as for plotting functions or defining evaluation grids.[15]

np.arange() is best reserved for cases with integer steps.

Python

```python
# Using arange with an integer step
arr_arange = np.arange(0, 10, 2)
print(arr_arange)
# Output: [0 2 4 6 8]

# Using linspace to get 5 points from 0 to 10 (inclusive)
arr_linspace = np.linspace(0, 10, 5)
print(arr_linspace)
# Output: [ 0.   2.5  5.   7.5 10. ]
```

The table below clarifies the distinction between np.arange() and np.linspace().

Feature	np.arange()	np.linspace()	Use Case Example
Primary Control	Step size between values (step) [18]	Total number of values (num) [18]	Creating an array of integers from 0 to 99: np.arange(100).
Endpoint Handling	stop value is excluded (half-open interval) [22]	stop value is included by default (closed interval) [3]	Generating 100 coordinates for a plot between 0 and 2π: np.linspace(0, 2*np.pi, 100).

Data Type Behavior	Output type inferred from inputs; can be imprecise with floats [16]	Typically creates float arrays to handle calculated spacing precisely [1]	Creating a sequence from 0 to 1 with a step of 0.1. linspace is safer: np.linspace(0, 1, 11).
Best For	Generating sequences with a fixed, integer-based increment.	Generating sequences for plotting, function evaluation, or when endpoint precision is critical.	Creating feature values that must be integers, like day of the week.

2.5 The Power of Randomness: The np.random Module

Random number generation is fundamental to many machine learning tasks, from initializing model weights to creating synthetic data or shuffling datasets for cross-validation. NumPy's np.random submodule is the workhorse for these tasks.[12]

- np.random.rand(d0, d1,...): Creates an array of a given shape with random values from a **uniform distribution** over the interval [0,1).[3]
- np.random.randn(d0, d1,...): Creates an array of a given shape with values from a **standard normal (Gaussian) distribution** (mean 0, variance 1). This is extremely common for initializing the weights of neural networks.[14]
- np.random.randint(low, high, size): Generates an array of random integers within the half-open interval [low, high).[4]

To ensure that experiments are reproducible, it is crucial to set a random seed using np.random.seed(). This function fixes the sequence of random numbers that NumPy will generate, ensuring that anyone who runs the code with the same seed will get the exact same "random" results.[13]

Python

```python
# Set a seed for reproducibility
np.random.seed(42)
```

```
# Create a 2x2 array of uniformly distributed random numbers
rand_arr = np.random.rand(2, 2)
print(rand_arr)
# Output:
# [[0.37454012 0.95071431]
#  [0.73199394 0.59865848]]

# Create a 2x2 array of normally distributed random numbers
randn_arr = np.random.randn(2, 2)
print(randn_arr)
# Output:
# [[ 0.15601864 -0.15241875]
#  [-0.03434575  1.46935877]]
```

Section 3: The Anatomy of an Array: Inspection and Manipulation

Once an array is created, the next step is to understand its structure and, if necessary, change it to fit the requirements of a machine learning model. The functions for inspecting and reshaping arrays are the fundamental tools of data preparation, or "data munging."

3.1 Inspecting Your Array's Properties

NumPy arrays have several key attributes that allow for quick inspection of their structure and metadata. These are essential for debugging and ensuring data is in the correct format.[17]

- .shape: Returns a tuple representing the size of the array in each dimension. For a 2D matrix, this is (rows, columns).[1]
- .ndim: Returns the number of dimensions (or axes) of the array.[13]
- .size: Returns the total number of elements in the array, which is the product of the dimensions in its shape.[4]
- .dtype: Returns the data type of the elements in the array.[1]

Python

```
arr = np.array([, ])

print(f"Shape: {arr.shape}")       # Output: Shape: (2, 3)
print(f"Dimensions: {arr.ndim}")  # Output: Dimensions: 2
print(f"Size: {arr.size}")        # Output: Size: 6
print(f"Data Type: {arr.dtype}") # Output: Data Type: int64
```

3.2 Reshaping Arrays: Changing the Structure without Changing the Data

Reshaping is one of the most common and critical operations in preparing data for machine learning. ML models often expect input data to have a very specific shape. For example, many Scikit-learn models expect a 2D array of shape (n_samples, n_features), while an image might initially be loaded as a 3D array of (height, width, color_channels).

The np.reshape() function (or the equivalent array method .reshape()) changes the shape of an array without changing its data, as long as the new shape is compatible with the original size.[1]

A powerful feature is the ability to use -1 for one of the dimensions. This tells NumPy to automatically calculate the correct size for that dimension based on the array's total size and the other specified dimensions.

Python

```
# Create a 1D array with 12 elements
arr_1d = np.arange(12)
print(arr_1d)
# Output: [ 0  1  2  3  4  5  6  7  8  9 10 11]

# Reshape it into a 3x4 matrix
arr_3x4 = arr_1d.reshape(3, 4)
```

```
print(arr_3x4)
# Output:
# [[ 0  1  2  3]
#  [ 4  5  6  7]
#  [ 8  9 10 11]]

# Reshape it into a 4x3 matrix using -1 for the number of rows
arr_4x3 = arr_1d.reshape(-1, 3)
print(arr_4x3.shape)
# Output: (4, 3)
```

3.3 Stacking and Splitting: Combining and Dividing Arrays

These operations are vital for assembling datasets from different sources or for separating features from target variables.

- **Concatenation and Stacking:** np.concatenate() joins a sequence of arrays along an existing axis.[1] However, the more intuitive functions np.vstack() (vertical stack) and np.hstack() (horizontal stack) are often preferred. vstack stacks arrays in sequence vertically (row-wise), while hstack stacks them horizontally (column-wise).[1] This is useful for adding new samples (rows) to a dataset or new features (columns).
- **Splitting:** Conversely, np.hsplit() and np.vsplit() can be used to split an array into multiple smaller arrays horizontally or vertically.[1] This is useful for separating a combined dataset back into its constituent parts, such as features and labels.

Python

```
a = np.array([, ])
b = np.array([, ])

# Vertical stacking (adding new rows)
v_stacked = np.vstack((a, b))
print(f"Vertically Stacked:\n{v_stacked}")
# Output:
# Vertically Stacked:
```

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

# Horizontal stacking (adding new columns)
h_stacked = np.hstack((a, b))
print(f"Horizontally Stacked:\n{h_stacked}")
# Output:
# Horizontally Stacked:
# [[1 1 3 3]
#  [2 2 4 4]]
```

These array manipulation functions are not just abstract tools; they are the verbs of data preparation. They form the bridge between raw data in various formats and the clean, structured (n_samples, n_features) matrix that most machine learning algorithms require. Mastering them is mastering the first practical step of any ML project.

3.4 Copies vs. Views: A Critical Gotcha

One of the most common sources of bugs for new NumPy users is the distinction between a copy and a view of an array. Misunderstanding this can lead to unintended modifications of data.

- **No Copy (Assignment):** A simple assignment like b = a does not create a new array. Both a and b now point to the exact same object in memory. Modifying b will also modify a.
- **View (Shallow Copy):** Slicing an array creates a **view**. A view is a new array object that looks at the *same underlying data* as the original array.[8] This is a memory-efficient practice, as it avoids duplicating large amounts of data. However, it means that if the view is modified, the original array will also change.
- **Copy (Deep Copy):** To create a completely independent array with its own data, the .copy() method must be used.[8] This creates a
  deep copy. Modifying the copy will have no effect on the original array. This is essential when an operation requires modifying a subset of data while preserving the original dataset.

Python

```python
# Original array
original = np.arange(5)
print(f"Original array: {original}")

# Create a view using a slice
view = original[1:4]
print(f"View: {view}")

# Modify an element in the view
view = 99
print(f"View after modification: {view}")
print(f"Original array after modifying view: {original}") # The original is also changed!
# Output:
# Original array: [0 1 2 3 4]
# View: [1 2 3]
# View after modification: [99  2  3]
# Original array after modifying view: [ 0 99  2  3  4]

# Create a deep copy
original = np.arange(5) # Reset original
copy = original[1:4].copy()
copy = 99 # Modify the copy
print(f"\nCopy after modification: {copy}")
print(f"Original array after modifying copy: {original}") # The original is unchanged
# Output:
# Copy after modification: [99  2  3]
# Original array after modifying copy: [0 1 2 3 4]
```

Section 4: Indexing and Slicing: Accessing Your Data with Precision

Accessing and manipulating specific subsets of data is a core task in data analysis and feature engineering. NumPy offers a powerful and flexible set of indexing techniques that go far beyond what is possible with standard Python lists.

4.1 Basic Indexing: The Fundamentals

Like Python lists, NumPy arrays use 0-based indexing, where the first element is at index 0.[1] For multidimensional arrays, indices are provided in a tuple, separated by commas.

Python

```
# 1D array
arr_1d = np.array()
print(f"Element at index 2: {arr_1d}")  # Output: Element at index 2: 30

# 2D array
arr_2d = np.array([, , ])
# Access the element at row 1, column 2
print(f"Element at (1, 2): {arr_2d}") # Output: Element at (1, 2): 6
```

4.2 Slicing: Accessing Sub-arrays

Slicing allows for the extraction of sub-arrays using the start:stop:step notation, which should be familiar to Python programmers. This syntax can be applied to each dimension of a NumPy array, making it a powerful tool for selecting rows, columns, or rectangular blocks of data.[13]

Python

```
arr_2d = np.array([, , ])
```

```
# Select the first two rows
print(f"First two rows:\n{arr_2d[0:2, :]}")
# Output:
# First two rows:
# [[1 2 3 4]
#  [5 6 7 8]]

# Select the third column
print(f"Third column:\n{arr_2d[:, 2]}")
# Output:
# Third column:
# [ 3  7 11]

# Select a 2x2 sub-matrix from the top-right corner
print(f"Top-right 2x2 sub-matrix:\n{arr_2d[0:2, 2:4]}")
# Output:
# Top-right 2x2 sub-matrix:
# [[3 4]
#  [7 8]]
```

4.3 Boolean Indexing: Filtering Your Data

Boolean indexing, also known as boolean masking, is one of NumPy's most powerful features for data selection. It allows for the filtering of array elements based on a condition, much like a WHERE clause in SQL.

The process involves two steps:

1. Create a boolean array (a "mask") by applying a conditional operator to the array. This results in an array of the same shape, containing True or False values.
2. Use this mask as the index for the original array. This returns a new array containing only the elements where the mask was True.[13]

Multiple conditions can be combined using the bitwise operators & (and) and | (or). Note that the standard Python and and or keywords do not work for element-wise comparison.[1]

Python

```
data = np.arange(10)
print(f"Original data: {data}")

# Create a boolean mask for values greater than 5
mask = data > 5
print(f"Boolean mask: {mask}")
# Output: Boolean mask:

# Apply the mask to select the elements
print(f"Values greater than 5: {data[mask]}")
# Output: Values greater than 5: [6 7 8 9]

# Combine conditions: select values that are even AND greater than 3
combined_mask = (data % 2 == 0) & (data > 3)
print(f"Even and >3: {data[combined_mask]}")
# Output: Even and >3: [4 6 8]
```

This paradigm of creating and applying masks is central to data analysis in NumPy. It provides a high-performance, vectorized way to perform sophisticated data queries and filtering operations directly in code.

4.4 Fancy Indexing: Accessing with Index Arrays

Fancy indexing allows for the selection of elements using an array of integer indices. This provides complete flexibility to select any combination of rows or specific elements in any order, which is not possible with simple slicing.[13]

Python

```
matrix = np.arange(16).reshape(4, 4)
print(f"Original matrix:\n{matrix}")
```

```
# Select rows 0 and 3
print(f"\nRows 0 and 3:\n{matrix[]}")
# Output:
# Rows 0 and 3:
# [[ 0  1  2  3]
#  [12 13 14 15]]

# Select elements at specific coordinates: (0, 1), (2, 3), and (3, 0)
coords_rows = np.array()
coords_cols = np.array()
print(f"\nElements at (0,1), (2,3), (3,0): {matrix[coords_rows, coords_cols]}")
# Output:
# Elements at (0,1), (2,3), (3,0): [ 1 11 12]
```

4.5 The np.nonzero() Function

As an alternative to boolean masking, np.nonzero() returns the *indices* of the elements that are not zero (or, more generally, satisfy a condition). It returns a tuple of arrays, one for each dimension, containing the indices of the non-zero elements.[1] This is useful when the positions of the elements are needed, rather than just their values.

Python

```
arr = np.array()

# Get the indices of non-zero elements
indices = np.nonzero(arr)
print(f"Indices of non-zero elements: {indices}")
# Output: Indices of non-zero elements: (array(),)

# Use these indices to retrieve the values (same as boolean indexing)
print(f"Values at those indices: {arr[indices]}")
# Output: Values at those indices: [ 5 10 15]
```

Section 5: The Computational Engine: NumPy Operations for ML

At the heart of NumPy's utility for machine learning is its suite of high-performance mathematical functions. These functions operate on entire arrays without the need for explicit Python loops, a concept known as vectorization.

5.1 Vectorization and Universal Functions (ufuncs): The "No-Loop" Philosophy

Vectorization is the practice of applying an operation to an entire array at once, rather than element by element.[4] This "no-loop" philosophy is central to writing efficient and readable NumPy code. Instead of writing a

for loop in Python, a single expression is used.

This is made possible by **Universal Functions**, or **ufuncs**. A ufunc is a function that operates on ndarrays in an element-wise fashion.[24] They are highly optimized wrappers around C code that perform the core computation. All standard arithmetic operators (

+, -, *, /, **) are convenient shorthands for their corresponding ufuncs (np.add, np.subtract, np.multiply, np.divide, np.power).[13]

Beyond basic arithmetic, NumPy provides a vast library of ufuncs essential for machine learning, including np.exp(), np.log(), np.sqrt(), and trigonometric functions like np.sin() and np.cos().[26]

Python

```python
arr = np.arange(1, 6)

# Instead of a Python loop:
# result =
# for x in arr:
#     result.append(x * 2)
# Use vectorization:
result = arr * 2
print(f"Vectorized multiplication: {result}")
# Output: Vectorized multiplication: [ 2  4  6  8 10]

# Applying a ufunc (np.exp)
exp_result = np.exp(arr)
print(f"Vectorized exponential: {exp_result}")
# Output: Vectorized exponential: [  2.71828183   7.3890561   20.08553692  54.59815003 148.4131591 ]
```

5.2 Broadcasting: The Magic of Mismatched Shapes

Broadcasting is the powerful mechanism that allows NumPy to perform arithmetic operations on arrays of different shapes.[3] It is one of NumPy's most important features, but it can be confusing at first.

The simplest case is an operation between an array and a scalar. The scalar value is "broadcast" or stretched to match the shape of the array, so the operation can be performed element-wise.[13]

Python

```
arr = np.array([, ])
# Add 100 to every element
result = arr + 100
print(result)
# Output:
# [[101 102 103]
#  [104 105 106]]
```

The general rules for broadcasting between two arrays are more complex:

1. If the arrays do not have the same number of dimensions, the shape of the one with fewer dimensions is padded with ones on its left side.
2. The two arrays are compatible in a dimension if they have the same size in that dimension, or if one of the arrays has a size of 1 in that dimension.

3. If these conditions are not met, a ValueError is raised.

A common machine learning application is standardizing a dataset. This involves subtracting the mean of each feature (a 1D vector) from all samples in the data matrix (a 2D matrix). Broadcasting handles this gracefully.

Python

```python
# A 3x3 data matrix (3 samples, 3 features)
X = np.array([,
              ,
              ])

# A 1D array of feature means
means = np.array()

# Subtract the means from X.
# The shape of means (3,) is broadcast to match X (3, 3).
# It behaves as if we subtracted [, , ].
X_centered = X - means
print(X_centered)
# Output:
# [[-3. -3. -3.]
#  [ 0.  0.  0.]
#  [ 3.  3.  3.]]
```

5.3 Aggregate Functions and the axis Parameter

Aggregate functions perform a computation on a set of values and return a single result. Common examples include np.sum(), np.mean(), np.std() (standard deviation), np.min(), and np.max().[23]

The most powerful aspect of these functions is the axis parameter, which specifies the dimension along which the aggregation should be performed.[30]

- axis=None (the default): The aggregation is performed on all elements of the

array, returning a single scalar value.
- axis=0: The operation is performed "down the columns." It collapses the rows, resulting in an aggregation for each column.
- axis=1: The operation is performed "across the rows." It collapses the columns, resulting in an aggregation for each row.

This is extremely useful in data analysis. For a dataset where rows are samples and columns are features, axis=0 is used to compute statistics for each feature, while axis=1 is used to compute statistics for each sample.

Python

```python
data = np.arange(1, 10).reshape(3, 3)
print(f"Data:\n{data}")

# Aggregate over the entire array (axis=None)
print(f"\nSum of all elements: {np.sum(data)}") # Output: 45

# Aggregate down the columns (axis=0)
# e.g., for column 0: 1 + 4 + 7 = 12
print(f"Sum of each column: {np.sum(data, axis=0)}") # Output: [12 15 18]

# Aggregate across the rows (axis=1)
# e.g., for row 0: 1 + 2 + 3 = 6
print(f"Sum of each row: {np.sum(data, axis=1)}") # Output: [ 6 15 24]
```

5.4 Linear Algebra: The Language of Machine Learning

Linear algebra is the mathematical foundation of most machine learning algorithms, and NumPy provides a comprehensive suite of functions to perform linear algebra operations efficiently.[4]

- **Dot Product:** The dot product is arguably the most important operation in machine learning. It is the core of weighted sums in linear models and neural networks.[33] The

np.dot() function and the @ operator (in Python 3.5+) are used to compute it.[33] The behavior of this function adapts based on the dimensions of the input arrays:
  - **Two 1D arrays (vectors):** It computes the scalar dot product (sum of element-wise products).
  - **Two 2D arrays (matrices):** It performs standard matrix multiplication. The inner dimensions must match (i.e., the number of columns in the first matrix must equal the number of rows in the second).
  - **Higher dimensions:** It performs a sum of products over specific axes.
- **Transpose:** Transposing a matrix (swapping its rows and columns) is a frequent operation. This is easily done using the .T attribute.[4]

The dot product is a polymorphic workhorse. A single neuron's output can be seen as np.dot(inputs, weights). A full layer's forward pass for one sample is a matrix-vector product, np.dot(weights_matrix, input_vector). A forward pass for an entire batch of data is a matrix-matrix product, np.dot(input_batch, weights_matrix.T). The fact that np.dot() and @ handle all these cases makes the code for ML models remarkably concise and powerful.[34]

Python

```python
# Dot product of two vectors
v1 = np.array()
v2 = np.array()
# 1*4 + 2*5 + 3*6 = 4 + 10 + 18 = 32
print(f"Vector dot product: {np.dot(v1, v2)}") # Output: 32

# Matrix multiplication
mat1 = np.array([, ])
mat2 = np.array([, ])
print(f"\nMatrix multiplication:\n{mat1 @ mat2}")
# Output:
# [[19 22]
#  [43 50]]

# Transpose
print(f"\nOriginal matrix:\n{mat1}")
print(f"Transposed matrix:\n{mat1.T}")
# Output:
```

```
# Original matrix:
# [[1 2]
#  [3 4]]
# Transposed matrix:
# [[1 3]
#  [2 4]]
```

Part II: Visualizing Data and Models - Matplotlib

While NumPy provides the computational engine, Matplotlib provides the visual canvas. Data visualization is not an optional extra in machine learning; it is a critical tool for exploratory data analysis (EDA), model debugging, and communicating results. This part of the report focuses on how to use Matplotlib to turn NumPy arrays into insightful plots.

Section 6: Introduction to Matplotlib: Painting a Picture of Your Data

Matplotlib is the most widely used and foundational plotting library in Python, providing the power to create a vast range of static, animated, and interactive visualizations.[36]

6.1 The Two Interfaces: Pyplot vs. Object-Oriented API

Matplotlib offers two distinct interfaces for creating plots, which can be a source of confusion for beginners.[37]

1. **The pyplot API:** This is a state-based interface that implicitly tracks the "current" figure and axes. Functions like plt.plot() and plt.title() operate on whatever plot is currently active. While it is convenient for very simple, quick plots, it can quickly become ambiguous and lead to confusing code when creating more complex visualizations with multiple plots.[37]
2. **The Object-Oriented (OO) API:** This is the recommended, more robust, and

more explicit approach. It involves creating explicit Figure and Axes objects and then calling methods directly on those objects. This makes the code clearer, more organized, and easier to maintain, especially for complex figures.[11]

This report will focus exclusively on the **Object-Oriented API**, as it represents the professional standard and is a crucial investment for anyone serious about data visualization. Committing to the OO API from the start, while slightly more verbose for a single plot, pays massive dividends in complexity management and code clarity. It prevents the "spaghetti code" that often results from the state-based pyplot API.

6.2 Anatomy of a Plot: Figure and Axes

The core of the OO API is understanding the two fundamental components of a visualization [11]:

- **Figure (fig):** This is the top-level container for all plot elements. It can be thought of as the entire canvas or window. A single Figure can contain one or more Axes.
- **Axes (ax):** This is the individual plot itself. It is the region where data is plotted and contains the x-axis, y-axis, title, grid, and other graphical elements. Most interactions will be with the Axes object.

The standard way to create these objects is with the plt.subplots() function. When called with no arguments, it creates a single Figure and a single Axes object.[11]

Python

```python
import matplotlib.pyplot as plt

# Create a Figure and a single Axes object
fig, ax = plt.subplots()

# Now, methods are called on the 'ax' object to build the plot
# (e.g., ax.plot(...), ax.set_title(...))
```

6.3 A Simple Workflow

A standard workflow for creating any plot with the OO API can be broken down into five steps [11]:

1. **Import libraries:** import matplotlib.pyplot as plt and import numpy as np.
2. **Prepare data:** Create or load the data, typically in NumPy arrays.
3. **Set up the plot:** Create the Figure and Axes objects: fig, ax = plt.subplots().
4. **Plot the data:** Call a plotting method on the Axes object, e.g., ax.plot(x, y).
5. **Customize and show:** Add titles, labels, legends, etc., and finally display the plot with plt.show().

Section 7: The Data Scientist's Toolkit: Essential Plot Types

The choice of plot is not arbitrary; it is driven by the type of data being analyzed and the specific question being asked. This section covers the four most essential plot types for machine learning analysis.

7.1 Line Plots (ax.plot)

- **Use Case:** A line plot is ideal for visualizing trends over a continuous interval, most commonly time. It is also the standard way to plot mathematical functions.[11] It answers the question, "How does a value change over a continuous range?"
- **Example:** Plotting a simple quadratic function.

Python

```python
import matplotlib.pyplot as plt
import numpy as np

# 1. Prepare data
```

```
x = np.linspace(-5, 5, 100)
y = x ** 2

# 2. Set up the plot
fig, ax = plt.subplots()

# 3. Plot the data
ax.plot(x, y)

# 4. Customize and show (details in next section)
ax.set_title("Quadratic Function")
ax.set_xlabel("x")
ax.set_ylabel("$x^2$")
ax.grid(True)
plt.show()
```

7.2 Scatter Plots (ax.scatter)

- **Use Case:** A scatter plot is fundamental for exploratory data analysis. It visualizes the relationship between two numerical variables, helping to identify correlations, clusters, patterns, and outliers.[41] It answers the question, "What is the relationship between two numerical features?"
- **Example:** Plotting two sets of random data that have a linear relationship.

```python
# 1. Prepare data
np.random.seed(42)
x = np.linspace(0, 10, 50)
y = 2 * x + 1 + np.random.randn(50) * 2 # Linear relationship with noise

# 2. Set up the plot
fig, ax = plt.subplots()

# 3. Plot the data
```

```python
ax.scatter(x, y)

# 4. Customize and show
ax.set_title("Scatter Plot of Correlated Data")
ax.set_xlabel("Feature 1")
ax.set_ylabel("Feature 2")
plt.show()
```

7.3 Histograms (ax.hist)

- **Use Case:** A histogram is used to understand the distribution of a single numerical variable. It groups data into "bins" and displays the frequency of observations in each bin, revealing the data's underlying probability distribution (e.g., normal, skewed, bimodal).[11] It answers the question, "How are the values of a single feature distributed?"
- **Example:** Visualizing a normal distribution.

Python

```python
# 1. Prepare data
data = np.random.randn(1000) # 1000 samples from a standard normal distribution

# 2. Set up the plot
fig, ax = plt.subplots()

# 3. Plot the data
# The 'bins' parameter controls the number of bars
ax.hist(data, bins=30, edgecolor='black')

# 4. Customize and show
ax.set_title("Histogram of a Normal Distribution")
ax.set_xlabel("Value")
ax.set_ylabel("Frequency")
plt.show()
```

7.4 Bar Charts (ax.bar)

- **Use Case:** A bar chart is used to compare a numerical value across a set of discrete categories.[11] It answers the question, "How does a numerical metric compare across several distinct groups?"
- **Example:** Comparing the accuracy of different hypothetical models.

Python

```python
# 1. Prepare data
models =
accuracies =

# 2. Set up the plot
fig, ax = plt.subplots()

# 3. Plot the data
ax.bar(models, accuracies, color=['skyblue', 'salmon', 'lightgreen'])

# 4. Customize and show
ax.set_title("Model Accuracy Comparison")
ax.set_xlabel("Model")
ax.set_ylabel("Accuracy (%)")
ax.set_ylim(0, 100) # Set y-axis limits
plt.show()
```

Section 8: From Plot to Publication: Customizing Your Visualizations

A raw plot conveys information, but a well-customized plot communicates an insight effectively. Customization is not about mere decoration; it is about guiding the viewer's attention and preventing misinterpretation.

8.1 Essential Labels: Titles and Axes

A plot without labels is ambiguous. The most basic and essential customizations are titles and axis labels.[36]

- ax.set_title(): Sets the main title for the Axes.
- ax.set_xlabel() and ax.set_ylabel(): Set the labels for the x and y axes, respectively.

8.2 Clarifying with Legends

When multiple datasets are plotted on the same Axes (e.g., comparing two models' performance over time), a legend is crucial to distinguish them. This is done by adding a label argument to each plotting call and then invoking ax.legend().[36]

8.3 Aesthetics: Colors, Markers, and Line Styles

Controlling the visual style of plotted elements is key to creating clear and professional-looking graphics.

- color: Can be set using names ('red'), abbreviations ('b'), or hex codes ('#FF5733').[39]
- marker: In line and scatter plots, this changes the style of the data points (e.g., 'o' for circles, 'x' for crosses, '^' for triangles).[41]
- linestyle: For line plots, this controls the line style (e.g., '-' for solid, '--' for dashed, ':' for dotted).[39]
- alpha: Sets the transparency of an element (from 0.0 for fully transparent to 1.0 for fully opaque), which is very useful for scatter plots with overlapping points.[41]

8.4 Sizing and Layout: figsize and subplots

Matplotlib provides full control over the figure's size and layout.

- figsize: The plt.subplots() function takes a figsize=(width, height) argument, specified in inches, to control the overall size of the figure.[39]
- **Multiple Subplots:** To create a grid of plots, pass the number of rows and columns to plt.subplots(). For example, fig, axes = plt.subplots(nrows=2, ncols=2) creates a 2x2 grid. axes is then a 2D NumPy array of Axes objects, which can be indexed (e.g., axes) to plot on a specific subplot.[11]

Python

```
# Example of a customized multi-plot figure
x = np.linspace(0, 10, 100)
y1 = np.sin(x)
y2 = np.cos(x)

# Create a figure with a specific size
fig, ax = plt.subplots(figsize=(8, 5))

# Plot two lines with labels, colors, and styles
ax.plot(x, y1, color='blue', linestyle='-', label='Sine')
ax.plot(x, y2, color='red', linestyle='--', label='Cosine')

# Add customizations
ax.set_title("Sine and Cosine Functions", fontsize=16)
ax.set_xlabel("x", fontsize=12)
ax.set_ylabel("f(x)", fontsize=12)
ax.grid(True)
ax.legend(loc='upper right') # Display the legend

plt.show()
```

Part III: Bridging Theory and Practice - NumPy & Matplotlib in

Machine Learning

This final part synthesizes the concepts from Parts I and II by applying them to solve common, practical machine learning visualization tasks. These examples demonstrate the powerful synergy between NumPy's computational capabilities and Matplotlib's visualization tools.

Section 9: Practical Application: Visualizing ML Concepts

9.1 Use Case 1: Plotting an Activation Function (Sigmoid)

Activation functions are a core component of neural networks, introducing non-linearity that allows models to learn complex patterns. The sigmoid function is a classic example. Visualizing it is a perfect "hello world" task for scientific plotting.

- **NumPy's Role:** np.linspace() is used to generate a smooth, evenly spaced range of input values. The sigmoid formula, $S(x)=1+e-x1$, is implemented efficiently using NumPy's vectorized np.exp() ufunc. This avoids a slow Python loop and computes the output for all x-values at once.[48]
- **Matplotlib's Role:** ax.plot() is used to draw the characteristic "S"-shaped curve. Customization functions add a title, axis labels, and a grid to make the plot clear and readable.[49]

Python

```python
import matplotlib.pyplot as plt
import numpy as np

def sigmoid(x):
    """Computes the sigmoid function."""
    return 1 / (1 + np.exp(-x))

# 1. Prepare data using NumPy
x_values = np.linspace(-10, 10, 200)
y_values = sigmoid(x_values)

# 2. Set up the plot
fig, ax = plt.subplots(figsize=(8, 5))

# 3. Plot the data
ax.plot(x_values, y_values, color='purple', linewidth=2)

# 4. Customize and show
ax.set_title("The Sigmoid Activation Function")
ax.set_xlabel("Input (x)")
ax.set_ylabel("Sigmoid Output (S(x))")
ax.grid(True)
ax.axhline(y=0.5, color='grey', linestyle=':') # Add horizontal line at y=0.5
ax.axvline(x=0, color='grey', linestyle=':') # Add vertical line at x=0
plt.show()
```

9.2 Use Case 2: Exploratory Data Analysis (EDA) on a Dataset

Before training any model, it is crucial to explore the dataset to understand its structure and the relationships between features. This process is known as Exploratory Data Analysis (EDA).

Here, a simple 2D dataset is simulated for a binary classification problem. Two "blobs" of data points are generated, each from a different normal distribution, representing two distinct classes.[52]

- **NumPy's Role:** np.random.randn() is used to create the two clusters of data points. np.vstack() and np.hstack() are used to assemble the final features matrix X and labels vector y.
- **Matplotlib's Role:**
  - ax.scatter() provides an immediate visualization of the feature space. The c (color) parameter is mapped to the class labels y, instantly revealing whether the classes are well-separated, overlapping, or complex. This is often the first

step in assessing if a simple linear model might be sufficient.[52]

- ax.hist() can be used to compare the distribution of a single feature across the two classes. Plotting two semi-transparent histograms on the same axes can reveal if a feature is discriminative.[52]

Python

```python
# 1. Prepare data using NumPy
np.random.seed(0)
# Class 0 data
X0 = np.random.randn(100, 2) + np.array()
y0 = np.zeros(100)
# Class 1 data
X1 = np.random.randn(100, 2) + np.array([-2, -2])
y1 = np.ones(100)

# Combine into a single dataset
X = np.vstack((X0, X1))
y = np.hstack((y0, y1))

# 2. Set up the plot
fig, ax = plt.subplots(figsize=(8, 6))

# 3. Plot the data
scatter = ax.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.RdYlBu, alpha=0.7, edgecolors='k')

# 4. Customize and show
ax.set_title("Feature Space of Simulated Data")
ax.set_xlabel("Feature 1")
ax.set_ylabel("Feature 2")
ax.legend(handles=scatter.legend_elements(), labels=['Class 0', 'Class 1'])
ax.grid(True)
plt.show()
```

9.3 Use Case 3: Visualizing a Classifier's Decision Boundary

This is a more advanced and powerful application that perfectly demonstrates the interplay between a machine learning model, NumPy, and Matplotlib. A decision boundary is the surface that separates the regions of the feature space where a model predicts different classes. Visualizing it shows how the model "sees" the data and makes its decisions.[53]

This visualization makes an abstract model's mathematical parameters (like the weights in logistic regression) tangible by showing their geometric behavior in the feature space. A straight boundary indicates a linear model, while a curved boundary indicates a non-linear one. A boundary that contorts to capture every outlier is a classic visual sign of overfitting.[54] It is a powerful diagnostic tool that moves beyond simple accuracy scores to provide a qualitative, intuitive understanding of a model's behavior.

The workflow is as follows:

1. **Train a Model:** A simple classifier (e.g., Logistic Regression from Scikit-learn) is trained on the 2D data.
2. **Create a Grid (NumPy):** np.meshgrid() is used to create a fine grid of points that covers the entire feature space. This involves defining the min/max range for each axis and generating coordinate matrices xx and yy.[53] These are then flattened and combined into a list of points to be fed to the model.
3. **Predict on the Grid (Model):** The trained model's .predict() method is called on every point in the grid.
4. **Plot the Boundary (Matplotlib):** ax.contourf() (filled contour plot) is used to color the background of the plot according to the model's predictions for each point on the grid. This creates the colored decision regions. The original training data is then overlaid with ax.scatter() to show how well the boundary separates the actual classes.[53]

Python

```python
from sklearn.linear_model import LogisticRegression

# Using the same data (X, y) from the previous example
# 1. Train a simple model
model = LogisticRegression()
model.fit(X, y)

# 2. Create a meshgrid with NumPy
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02),
                     np.arange(y_min, y_max, 0.02))

# 3. Predict on every point of the grid
grid_points = np.c_[xx.ravel(), yy.ravel()]
Z = model.predict(grid_points)
Z = Z.reshape(xx.shape)

# 4. Plot the decision boundary and data points with Matplotlib
fig, ax = plt.subplots(figsize=(8, 6))
ax.contourf(xx, yy, Z, cmap=plt.cm.RdYlBu, alpha=0.8)

# Plot the training points
scatter = ax.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.RdYlBu, edgecolors='k')

ax.set_title("Logistic Regression Decision Boundary")
ax.set_xlabel("Feature 1")
ax.set_ylabel("Feature 2")
ax.legend(handles=scatter.legend_elements(), labels=['Class 0', 'Class 1'])
plt.show()
```

Section 10: Conclusion and Your Path Forward

This report has provided a comprehensive, machine-learning-focused introduction to the NumPy and Matplotlib libraries. The journey has progressed from the fundamental reasons for NumPy's existence to the practical application of both libraries in visualizing complex machine learning concepts.

10.1 Summary of Core Concepts

The core takeaways are twofold. First, **NumPy is the high-performance computational engine** for machine learning in Python. Its ndarray object, with its contiguous memory layout and homogeneous data type, enables vectorized operations via ufuncs that are orders of magnitude faster than standard Python. Mastering array creation, manipulation, indexing, and linear algebra with NumPy is a non-negotiable skill.

Second, **Matplotlib is the primary tool for visual data exploration and model analysis**. Adopting the object-oriented API (fig, ax = plt.subplots()) provides a robust and scalable framework for creating everything from simple line plots to complex, multi-layered visualizations like decision boundaries. Customization is not just for aesthetics; it is for effective communication of results.

10.2 Next Steps in the ML Ecosystem

The knowledge gained from this report serves as the essential foundation for exploring the broader machine learning ecosystem. The logical next steps include:

- **Pandas:** A library built on NumPy that provides the DataFrame object, a powerful tool for working with labeled, tabular data. It simplifies tasks like loading data from files (e.g., CSVs), handling missing values, and performing complex data manipulations.[11]
- **Scikit-learn:** The preeminent library for classical machine learning in Python. It uses NumPy arrays as its primary data format for inputs (X) and targets (y). The skills learned here are directly applicable to preparing data for Scikit-learn models and using Matplotlib to evaluate their performance.[11]
- **Seaborn:** A high-level statistical plotting library built on top of Matplotlib. It simplifies the creation of sophisticated and aesthetically pleasing statistical plots (like heatmaps, violin plots, and pair plots) with less code.[38]

10.3 Final Encouragement

By mastering the fundamentals of NumPy and Matplotlib presented in this report, a new practitioner builds the computational and visual literacy required to confidently approach a wide range of machine learning challenges. The ability to efficiently manipulate numerical data and to visualize it effectively are the twin pillars upon which successful data science projects are built. The path forward involves applying these foundational tools to real-world datasets and more advanced algorithms, a journey for which this guide has prepared the groundwork.

Works cited

1. the absolute basics for beginners — NumPy v2.4.dev0 Manual, accessed on July 25, 2025, https://numpy.org/devdocs//user/absolute_beginners.html
2. Beginner's Guide to Python Numpy for AI/ML: Basics, Arrays, and Examples - Medium, accessed on July 25, 2025, https://medium.com/@expertappdevs/beginners-guide-to-python-numpy-for-ai-ml-basics-arrays-and-examples-da67f8d74579
3. The Ultimate NumPy Tutorial for Data Science Beginners - Analytics Vidhya, accessed on July 25, 2025, https://www.analyticsvidhya.com/blog/2020/04/the-ultimate-numpy-tutorial-for-data-science-beginners/
4. Everything You Need to Know About NumPy for Machine Learning - Comet, accessed on July 25, 2025, https://www.comet.com/site/blog/everything-you-need-to-know-about-numpy-for-machine-learning/
5. Python Lists VS Numpy Arrays - GeeksforGeeks, accessed on July 25, 2025, https://www.geeksforgeeks.org/python/python-lists-vs-numpy-arrays/
6. Python Lists Vs. NumPy Arrays: A Deep Dive into Memory Layout and Performance Benefits, accessed on July 25, 2025, https://www.dataleadsfuture.com/python-lists-vs-numpy-arrays-a-deep-dive-into-memory-layout-and-performance-benefits/
7. Numpy vs Traditional Python Lists: A Performance Showdown | by veyak - Medium, accessed on July 25, 2025, https://medium.com/@vakgul/numpy-vs-traditional-python-lists-a-performance-showdown-1e8bebc55933
8. the absolute basics for beginners — NumPy v2.4.dev0 Manual, accessed on July 25, 2025, https://numpy.org/devdocs/user/absolute_beginners.html
9. Why is NumPy Much Faster Than Lists? - Reddit, accessed on July 25, 2025, https://www.reddit.com/r/Numpy/comments/1cz52op/why_is_numpy_much_faster_than_lists/
10. NumPy vs Python Lists – Who's Faster? (Speed Test with 10 Million Elements) - YouTube, accessed on July 25, 2025, https://www.youtube.com/watch?v=YYMaMTW-2hw
11. Beginner's Guide To Matplotlib (With Code Examples) | Zero To ..., accessed on July 25, 2025, https://zerotomastery.io/blog/matplotlib-guide-python/

12. NumPy Tutorial - Python Library - GeeksforGeeks, accessed on July 25, 2025, https://www.geeksforgeeks.org/python/numpy-tutorial/
13. NumPy for Machine Learning - Made With ML by Anyscale, accessed on July 25, 2025, https://madewithml.com/courses/foundations/numpy/
14. Different Ways to Create Numpy Arrays in Python - GeeksforGeeks, accessed on July 25, 2025, https://www.geeksforgeeks.org/different-ways-to-create-numpy-arrays-in-python/
15. NumPy quickstart — NumPy v2.4.dev0 Manual, accessed on July 25, 2025, https://numpy.org/devdocs/user/quickstart.html
16. Array creation — NumPy v2.4.dev0 Manual, accessed on July 25, 2025, https://numpy.org/devdocs//user/basics.creation.html
17. Creating and Using NumPy Arrays - A Complete Guide - Codecademy, accessed on July 25, 2025, https://www.codecademy.com/article/numpy-arrays-guide
18. NumPy: The Difference Between np.linspace and np.arange - Statology, accessed on July 25, 2025, https://www.statology.org/numpy-linspace-vs-arange/
19. Generate equally-spaced values including the right end using NumPy.arange, accessed on July 25, 2025, https://stackoverflow.com/questions/59777735/generate-equally-spaced-values-including-the-right-end-using-numpy-arange
20. What is the difference between np.linspace and np.arange? - Stack Overflow, accessed on July 25, 2025, https://stackoverflow.com/questions/62106028/what-is-the-difference-between-np-linspace-and-np-arange
21. NumPy: arange() and linspace() to generate evenly spaced values | note.nkmk.me, accessed on July 25, 2025, https://note.nkmk.me/en/python-numpy-arange-linspace/
22. numpy.arange — NumPy v2.3 Manual, accessed on July 25, 2025, https://numpy.org/doc/stable/reference/generated/numpy.arange.html
23. Useful NumPy functions in Data Science | by Akriti Upadhyay - Medium, accessed on July 25, 2025, https://medium.com/@akriti.upadhyay/useful-numpy-functions-in-data-science-dc078f7cd36b
24. Universal functions (ufunc) basics — NumPy v2.2 Manual, accessed on July 25, 2025, https://numpy.org/doc/stable/user/basics.ufuncs.html
25. NumPy ufuncs | Universal functions - GeeksforGeeks, accessed on July 25, 2025, https://www.geeksforgeeks.org/python/numpy-ufunc/
26. What are Ufuncs in NumPy - Codecademy, accessed on July 25, 2025, https://www.codecademy.com/article/what-are-ufuncs-in-numpy
27. Universal Functions (ufuncs) in NumPy - Cybrosys Technologies, accessed on July 25, 2025, https://www.cybrosys.com/blog/universal-functions-in-numpy
28. Mathematical and Statistical functions on NumPy Arrays | Automated hands-on - CloudxLab, accessed on July 25, 2025, https://cloudxlab.com/assessment/displayslide/2509/numpy-mathematical-and-statistical-functions-on-numpy-arrays

29. Basic Statistical Analysis with NumPy - MachineLearningMastery.com, accessed on July 25, 2025, https://machinelearningmastery.com/basic-statistical-analysis-with-numpy/
30. numpy.sum — NumPy v2.3 Manual, accessed on July 25, 2025, https://numpy.org/doc/stable/reference/generated/numpy.sum.html
31. Numpy Aggregation Function - Medium, accessed on July 25, 2025, https://medium.com/@dsshiva/numpy-aggregation-function-0d4968b8e0cb
32. Understanding numpy.sum() with Axis Parameter | by why amit - Medium, accessed on July 25, 2025, https://medium.com/@whyamit101/understanding-numpy-sum-with-axis-parameter-1fc543fe9fa2
33. Dot Product - LearnDataSci, accessed on July 25, 2025, https://www.learndatasci.com/glossary/dot-product/
34. What is numpy.dot and When to Use It? | by why amit - Medium, accessed on July 25, 2025, https://medium.com/@whyamit101/what-is-numpy-dot-and-when-to-use-it-6e47be7bbf2d
35. NumPy dot() - DataCamp, accessed on July 25, 2025, https://www.datacamp.com/doc/numpy/dot
36. Matplotlib Tutorial - GeeksforGeeks, accessed on July 25, 2025, https://www.geeksforgeeks.org/python/matplotlib-tutorial/
37. Matplotlib Tutorial - Tutorialspoint, accessed on July 25, 2025, https://www.tutorialspoint.com/matplotlib/index.htm
38. Matplotlib — Visualization with Python, accessed on July 25, 2025, https://matplotlib.org/
39. Introduction to NumPy Matplotlib for Beginners - DataFlair, accessed on July 25, 2025, https://data-flair.training/blogs/numpy-matplotlib-tutorial/
40. Introduction to Plotting with Matplotlib in Python | DataCamp, accessed on July 25, 2025, https://www.datacamp.com/tutorial/matplotlib-tutorial-python
41. Python:Matplotlib .scatter() - pyplot - Codecademy, accessed on July 25, 2025, https://www.codecademy.com/resources/docs/matplotlib/pyplot/scatter
42. Matplotlib Scatter - GeeksforGeeks, accessed on July 25, 2025, https://www.geeksforgeeks.org/python/matplotlib-pyplot-scatter-in-python/
43. How to Create a Matplotlib Histogram? - StrataScratch, accessed on July 25, 2025, https://www.stratascratch.com/blog/how-to-create-a-matplotlib-histogram/
44. Working with Numpy Arrays and Data Visualization with Matplotlib | by Dev Nayak | Medium, accessed on July 25, 2025, https://medium.com/@davedesktop1010/working-with-numpy-arrays-and-data-visualization-with-matplotlib-4bb43a329936
45. Basic Barplot using Matplotlib - Python Graph Gallery, accessed on July 25, 2025, https://python-graph-gallery.com/1-basic-barplot/
46. Python Library 101: Pandas, Numpy and Matplotlib for Machine Learning or Data Science, accessed on July 25, 2025, https://www.youtube.com/watch?v=NZ0laizc4ug

47. Scatter plot in matplotlib - python charts, accessed on July 25, 2025, https://python-charts.com/correlation/scatter-plot-matplotlib/
48. Sigmoid Function in Machine Learning with Python.md - GitHub, accessed on July 25, 2025, https://github.com/xbeat/Machine-Learning/blob/main/Sigmoid%20Function%20in%20Machine%20Learning%20with%20Python.md
49. Understanding the Sigmoid Function and Its Use in NumPy | by Hey Amit - Medium, accessed on July 25, 2025, https://medium.com/@heyamit10/understanding-the-sigmoid-function-and-its-use-in-numpy-d99367850b92
50. Basic Numpy for Neural Networks - The Cloistered Monkey, accessed on July 25, 2025, https://necromuralist.github.io/Neurotic-Networking/posts/first-course/basic-numpy-for-neural-networks/index.html
51. A Python code that can reproduce the graph of the sigmoid function. | by Omar - Medium, accessed on July 25, 2025, https://medium.com/@bouimouass.o/a-python-code-that-can-reproduce-the-graph-of-the-sigmoid-function-51004981b805
52. Matplotlib for Machine Learning - The Data Frog, accessed on July 25, 2025, https://thedatafrog.com/en/articles/matplotlib-machine-learning/
53. Plotting Decision Boundaries using Numpy and Matplotlib | by Prakhar S - Medium, accessed on July 25, 2025, https://psrivasin.medium.com/plotting-decision-boundaries-using-numpy-and-matplotlib-f5613d8acd19
54. Plot Decision Boundary in Logistic Regression: Python Example - Analytics Yogi, accessed on July 25, 2025, https://vitalflux.com/plot-decision-boundary-using-logistic-regression-python-example/
55. How To Plot A Decision Boundary For Machine Learning Algorithms in Python | HackerNoon, accessed on July 25, 2025, https://hackernoon.com/how-to-plot-a-decision-boundary-for-machine-learning-algorithms-in-python-3o1n3w07
56. Plotting a decision boundary separating 2 classes using Matplotlib's pyplot - Stack Overflow, accessed on July 25, 2025, https://stackoverflow.com/questions/22294241/plotting-a-decision-boundary-separating-2-classes-using-matplotlibs-pyplot
57. Plotting a decision boundary python (give a good idea of how contourf matplotlib function works) - Stack Overflow, accessed on July 25, 2025, https://stackoverflow.com/questions/37429468/plotting-a-decision-boundary-python-give-a-good-idea-of-how-contourf-matplotli
58. Introduction to NumPy, Pandas and Matplotlib - Kaggle, accessed on July 25, 2025, https://www.kaggle.com/code/chats351/introduction-to-numpy-pandas-and-matplotlib
59. Guide to NumPy, pandas, and Data Visualization - Dataquest, accessed on July

25, 2025,
https://www.dataquest.io/guide/numpy-pandas-and-data-visualization-tutorial/
60. DecisionBoundaryDisplay — scikit-learn 1.7.1 documentation, accessed on July 25, 2025,
https://scikit-learn.org/stable/modules/generated/sklearn.inspection.DecisionBoundaryDisplay.html
61. Visualising ML DataSet Through Seaborn Plots and Matplotlib - GeeksforGeeks, accessed on July 25, 2025,
https://www.geeksforgeeks.org/machine-learning/visualising-ml-dataset-through-seaborn-plots-and-matplotlib/