Chapter 2: Python and Libraries

Python Basics

1. Variables and Data Types

- Variables are used to store data values
- Python has several built-in data types:
  - Numeric types: int, float, complex
  - Sequence types: list, tuple, range
  - Text type: str
  - Mapping type: dict
  - Set types: set, frozenset
  - Boolean type: bool
- Variables are created using the assignment operator (=)

Examples:

```
x = 5          # int
y = 3.14       # float
name = "John" # str
is_student = True  # bool
```

2. Operators

- Arithmetic operators: +, -, *, /, %, //, **
- Comparison operators: ==, !=, <, >, <=, >=
- Logical operators: and, or, not
- Assignment operators: =, +=, -=, *=, /=, %=, //=, **=

Examples:

```
x = 5
y = 3
print(x + y)  # 8
print(x > y)  # True
print((x > 3) and (y < 5))  # True
x += 2  # x is now 7
```

```
8
True
True
```

3. Control Flow

- if, elif, else statements for conditional execution
- for and while loops for iteration
- break, continue, and pass statements for loop control

Examples:

```
x = 5
if x > 0:
    print("Positive")
elif x < 0:
    print("Negative")
else:
    print("Zero")
for i in range(5):
    print(i)
i = 0
while i < 5:
    print(i)
    i += 1
```

```
Positive
0
1
```

```
2
3
4
0
1
2
3
4
```

4. Functions

- Functions are defined using the def keyword
- They can take parameters and return values

Examples:

```
def greet(name):
    print(f"Hello, {name}!")
def add(x, y):
    return x + y
greet("John")  # Hello, John!
result = add(3, 5)  # result is 8
```

```
Hello, John!
```

Python Data Structures

1. Lists

- Lists are ordered, mutable sequences
- They are created using square brackets []
- Elements can be accessed by index

Examples:

```
fruits = ["apple", "banana", "cherry"]
print(fruits[0])  # apple
```

```python
fruits[1] = "blueberry"
fruits.append("date")
print(fruits)  # ["apple", "blueberry", "cherry", "date"]

# Creating a list of squares of numbers from 1 to 5 using list comprehension
squares = [i ** 2 for i in range(1, 6)]
print(squares)  # Output: [1, 4, 9, 16, 25]

# Filtering even numbers from a list using list comprehension
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers = [num for num in numbers if num % 2 == 0]
print(even_numbers)  # Output: [2, 4, 6, 8, 10]

# Creating a list of strings using list comprehension
names = ["Alice", "Bob", "Charlie", "David"]
upper_case_names = [name.upper() for name in names]
print(upper_case_names)  # Output: ['ALICE', 'BOB', 'CHARLIE', 'DAVID']

# Rotating names
rotated_names = [names[i:]+names[:i] for i in range(4)]
print(rotated_names)

# More advanced examples
# Creating a 4x4 identity matrix using list comprehension
identity_matrix = [[1 if i == j else 0 for j in range(4)] for i in range(4)]
print(identity_matrix)
# Output: [[1, 0, 0, 0],
#          [0, 1, 0, 0],
#          [0, 0, 1, 0],
#          [0, 0, 0, 1]]
```

```
apple
['apple', 'blueberry', 'cherry', 'date']
[1, 4, 9, 16, 25]
[2, 4, 6, 8, 10]
['ALICE', 'BOB', 'CHARLIE', 'DAVID']
[['Alice', 'Bob', 'Charlie', 'David'], ['Bob', 'Charlie', 'David', 'Alice'], ['Charlie', 'Dav
[[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]]
```

```
# list all pdf files in the current directory
import glob
all_files = glob.glob('./*')
# print(all_files)

pdf_files = [file for file in all_files if file[-4:].lower()=='.pdf']
print(pdf_files)
```

```
['./ch03_Pytorch_Basics.pdf', './ch02_Python_and_Libraries.pdf', './tmp02_beamer.pdf', './ch
```

Iterating over a list

```
# Generating a list of random numbers and squaring even numbers
import random

random_numbers = [random.randint(1, 10) for _ in range(10)]
squared_even_numbers = [num ** 2 for num in random_numbers if num % 2 == 0]
print(random_numbers)
print(squared_even_numbers)
# Example output: [7, 9, 1, 3, 2, 10, 6, 8, 4, 5]
#                 [4, 100, 4, 36]


my_list = ['apple', 'banana', 'cherry', 'date']
# Using a while loop to iterate over the list
index = 0
while index < len(my_list):
    print(my_list[index])
    index += 1

# Output:
# 1
# 2
# 3
# 4
# 5

# Using a for loop with index
for index in range(len(my_list)):
    print(my_list[index])
```

```
# Output:
# 1
# 2
# 3
# 4
# 5

fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)

# Output:
# apple
# banana
# cherry

my_list = [1, 2, 3, 4, 5]

# Using a for loop to iterate over the list
for element in my_list:
    print(element)

# Using enumerate() to iterate over the list with index and value
for index, value in enumerate(my_list):
    print(f"Index {index}: {value}")

# Output:
# Index 0: apple
# Index 1: banana
# Index 2: cherry
# Index 3: date
```

```
[10, 9, 4, 3, 10, 10, 1, 2, 6, 5]
[100, 16, 100, 100, 4, 36]
apple
banana
cherry
date
apple
banana
cherry
date
```

```
apple
banana
cherry
1
2
3
4
5
Index 0: 1
Index 1: 2
Index 2: 3
Index 3: 4
Index 4: 5
```

Lists and Generators

Generators in Python are a memory-efficient way to create iterators. They use the `yield` keyword to produce values on-demand, conserving memory for large or infinite sequences. Generators are defined like functions but use `yield` to generate values dynamically. They are useful for lazy evaluation, infinite sequences, and streaming data.

Here's an extended version of the previous example that includes the use of the `iter()` and `next()` functions, as well as a generator function:

```python
# List comprehension to square numbers from 1 to 5
squared_numbers = [i ** 2 for i in range(1, 6)]
print(squared_numbers)  # Output: [1, 4, 9, 16, 25]

# Generator expression to square numbers from 1 to 5
squared_numbers_generator = (i ** 2 for i in range(1, 6))

# Using iter() and next() with the generator expression
generator_iterator = iter(squared_numbers_generator)
print(next(generator_iterator))  # Output: 1
print(next(generator_iterator))  # Output: 4
print(next(generator_iterator))  # Output: 9
print(next(generator_iterator))  # Output: 16
print(next(generator_iterator))  # Output: 25

# Generator function to square numbers from 1 to n
def square_generator(n):
    for i in range(1, n + 1):
```

```
        yield i ** 2
```

```
[1, 4, 9, 16, 25]
1
4
9
16
25
```

```
iter1 = iter(square_generator(5))
```

```
print(next(iter1))
```

```
1
```

```
# Using the generator function
generator_from_function = square_generator(5)
for squared_number in generator_from_function:
    print(squared_number)
# Output:
# 1
# 4
# 9
# 16
# 25
```

```
1
4
9
16
25
```

In this extended example, we first use the `iter()` function to obtain an iterator from the generator expression `squared_numbers_generator`. Then, we use the `next()` function to retrieve the next value from the iterator. We can call `next()` multiple times to get the next values from the generator expression.

Additionally, we introduce a generator function `square_generator(n)` that takes an argument `n` and yields the squares of numbers from 1 to `n`. We create a generator object

`generator_from_function` by calling the generator function with an argument of 5. We then iterate over the generator object using a `for` loop, which automatically handles calling `next()` and stopping at the `StopIteration` exception.

2. Tuples

- Tuples are ordered, immutable sequences
- They are created using parentheses ()
- Elements can be accessed by index

Examples:

```
point = (3, 5)
print(point[0])  # 3
x, y = point  # unpacking
```

```
3
```

3. Dictionaries

- Dictionaries are unordered, mutable mappings of keys to values
- They are created using curly braces {}
- Elements are accessed by key

Examples:

```
person = {"name": "John", "age":15, "city": "New York"}
print(person["name"])  # John
person["age"] = 31
person["email"] = "john@example.com"
print(person)  # {"name": "John", "age": 31, "city": "New York", "email": "john@example.co
```

```
John
{'name': 'John', 'age': 31, 'city': 'New York', 'email': 'john@example.com'}
```

4. Sets

- Sets are unordered, mutable collections of unique elements
- They are created using curly braces {} or the set() function

Examples:

```
fruits = {"apple", "banana", "cherry"}
fruits.add("date")
fruits.remove("banana")
print(fruits)  # {"apple", "cherry", "date"}
```

```
{'date', 'cherry', 'apple'}
```

Python Classes

Classes in Python provide a way to define custom data structures that encapsulate data (attributes) and functions (methods) that operate on that data. They are a fundamental concept in object-oriented programming (OOP) and are extensively used in PyTorch for defining neural network models, custom datasets, and more.

Defining a Class

A class is defined using the `class` keyword. The class name is a string that identifies the class. Class names typically follow the CamelCase convention. The class definition must be preceded by the `class` keyword, followed by the class name, and a colon. The class definition is followed by a series of class attributes and methods, which are defined using the `def` keyword. Class attributes are variables that are shared by all instances of the class. They are defined within the class but outside any methods. Class methods are functions that are defined within the class.

```
class MyClass:
# Class definition goes here
    pass
```

Class Attributes

Class attributes are variables that are shared by all instances of the class. They are defined within the class but outside any methods.

```
class MyClass:
    class_attr = 42
```

```
print(MyClass.class_attr)
```

```
42
```

Instance Attributes

Instance attributes are unique to each instance of the class. They are typically defined in the `__init__` method, which is a special method called when a new instance of the class is created.

```
class MyClass:
    def __init__(self, value):
        self.instance_attr = value
```

```
instance1 = MyClass(15)   # calls __init__
```

```
instance1.instance_attr
```

```
15
```

The `self` parameter refers to the instance being created and is used to access its attributes and methods.

Methods

Methods are functions defined within a class that operate on the class's data. They take `self` as the first parameter, which allows them to access the instance's attributes.

```
class MyClass:
    def __init__(self, value):
```

```
        self.instance_attr = value

    def my_method(self):
        print(f"Instance attribute: {self.instance_attr}")
```

```
instance2 = MyClass(4)
instance2.my_method()
```

```
Instance attribute: 4
```

Inheritance

Inheritance allows a class to inherit attributes and methods from another class, called the superclass or base class. The inheriting class is called the subclass or derived class.

```
class BaseClass:
    def base_method(self):
        print("This is the base method.")
class DerivedClass(BaseClass):
    def derived_method(self):
        print("This is the derived method.")

instance3 = DerivedClass()
instance3.base_method()
```

```
This is the base method.
```

The derived class inherits all attributes and methods from the base class and can also define its own.

Super()

The `super()` function allows a subclass to call methods from its superclass, even if the method has been overridden in the subclass.

```
class BaseClass:
    def __init__(self, value):
        self.value = value
class DerivedClass(BaseClass):
```

```
    def __init__(self, value, additional_value):
        super().__init__(value)
        self.additional_value = additional_value
```

```
class Father:
    def __init__(self, father_name):
        self.father_name = father_name
        print(f"Father's name: {self.father_name}")

class Mother:
    def __init__(self, mother_name):
        self.mother_name = mother_name
        print(f"Mother's name: {self.mother_name}")

class Child(Father, Mother):
    def __init__(self, father_name, mother_name, child_name):
        super().__init__(father_name)  # Initializes Father class
        super(Father, self).__init__(mother_name)  # Initializes Mother class
        self.child_name = child_name
        print(f"Child's name: {self.child_name}")
```

Abstract Base Classes

Abstract base classes (ABCs) define a common interface for a set of subclasses. They cannot be instantiated directly and may contain abstract methods that must be implemented by the subclasses.

```
from abc import ABC, abstractmethod
class MyABC(ABC):
    @abstractmethod
    def my_abstract_method(self):
        pass
class MyConcreteClass(MyABC):
    def my_abstract_method(self):
        print("Implementation of the abstract method.")
```

Example: PyTorch Model Definition

In PyTorch, neural network models are defined as classes that inherit from the `nn.Module` base class. The model's layers are defined as instance attributes in the `__init__` method, and

the forward pass is defined in the `forward` method.

```
import torch
import torch.nn as nn

class MyModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(MyModel, self).__init__()
        self.hidden = nn.Linear(input_size, hidden_size)
        self.output = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = torch.relu(self.hidden(x))
        x = self.output(x)
        return x
```

```
myModel = MyModel(2,3,1)
```

```
sample_input= torch.Tensor([20,30])
```

```
myModel(sample_input)
```

```
tensor([-0.5038], grad_fn=<ViewBackward0>)
```

This example defines a simple feedforward neural network with one hidden layer and ReLU activation.

Example: PyTorch Custom Dataset

PyTorch datasets are defined as classes that inherit from the `torch.utils.data.Dataset` base class. The dataset class must implement the `__len__` method, which returns the size of the dataset, and the `__getitem__` method, which returns a single sample from the dataset.

```
import torch
from torch.utils.data import Dataset
class MyDataset(Dataset):
    def __init__(self, data, labels):
        self.data = data
        self.labels = labels
```

```
    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        return self.data[idx], self.labels[idx]
```

This example defines a simple dataset that wraps a list of data points and their corresponding labels.

Python Libraries

1. NumPy

- NumPy, short for Numerical Python, is a fundamental library for scientific computing and data analysis in Python. It provides powerful tools for working with multi-dimensional arrays and matrices, along with a vast collection of mathematical functions.

1. Installation

Before embarking on your NumPy journey, ensure you have Python installed (preferably Python 3

``` bash
pip install numpy
```

If you're using a Conda environment, use:

``` bash
conda install numpy
```

2. NumPy Arrays

NumPy's core data structure is the `ndarray`, a multi-dimensional array capable of holding el

Creating Arrays:

- From Python lists:

```
import numpy as np

arr = np.array([1, 2, 3, 4])  # Creates a 1D array
```

- Using built-in functions:

```
zeros_arr = np.zeros((2, 3))  # Creates a 2x3 array filled with zeros
ones_arr = np.ones((4, 4))    # Creates a 4x4 array filled with ones
range_arr = np.arange(10)     # Creates an array with values from 0 to 9
```

Array Attributes:

- `ndim`: Number of dimensions (e.g., 2 for a matrix)
- `shape`: Tuple representing the size of each dimension (e.g., (2, 3) for a 2x3 matrix)
- `size`: Total number of elements in the array
- `dtype`: Data type of the elements (e.g., int32, float64)

3. Array Indexing and Slicing

Indexing:

Access individual elements using square brackets and indices:

````
::: {.cell execution_count=34}
``` {.python .cell-code}
first_element = arr[0]        # Access the first element
element_2d = zeros_arr[1, 2]  # Access element at row 1, column 2
last_element = arr[-1]       # Access the last element using -1
```

:::
````

Slicing:

Extract subarrays using slicing syntax:

```
sub_arr = arr[1:4]           # Elements from index 1 (inclusive) to 4 (exclusive)
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
all_rows_column_1 = arr2d[:, 1]  # All elements in the second column
```

4. Shape and Reshaping

Reshaping:

Change the shape of an array without modifying its data:

```
arr = np.arange(12)
new_arr = arr.reshape(3, 4)  # Reshape into a 3x4 matrix
```

5. Mathematical Operations

NumPy offers a wide range of mathematical functions:

Element-wise Operations:

```
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

c = a + b  # Element-wise addition
d = a * b  # Element-wise multiplication
```

Matrix Operations:

```
matrix_product = np.dot(a, b)  # Matrix multiplication
```

or equivalently,

```
matrix_product = a @ b  # Matrix multiplication, equivalent to np.dot(a, b) in this case
```

Note that `@` is specifically for matrix multiplication, while `np.dot()` can be used for both matrix multiplication and dot product, depending on the input arrays. For 2-D arrays, it is generally preferred to use `@` or `np.matmul()` over `np.dot()`.

6. Broadcasting

Broadcasting enables operations between arrays of different shapes:

```
arr = np.ones((3, 3))
arr + 5  # Adds 5 to each element of the array
```

```
array([[6., 6., 6.],
       [6., 6., 6.],
       [6., 6., 6.]])
```

7. Data Types and Precision

NumPy supports various data types, each with different precision and memory requirements:

```
int_arr = np.array([1, 2, 3], dtype='int16')  # Array of 16-bit integers
float_arr = np.array([1.5, 2.2, 3.8], dtype='float32')  # Array of 32-bit floats
```

8. Vectorized Computation

NumPy excels at performing operations on entire arrays without explicit loops, leading to significant performance gains:

````
::: {.cell execution_count=42}
``` {.python .cell-code}
arr = np.arange(10)
arr_squared = arr ** 2  # Squares each element in the array
````

:::

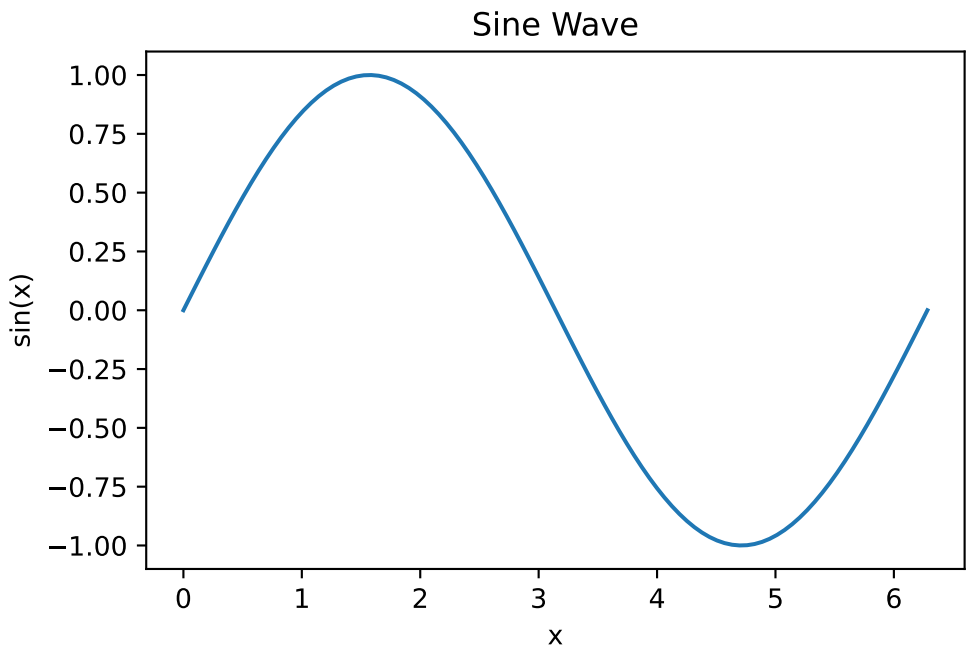
By mastering these core concepts, you'll be well-equipped to leverage NumPy's power for efficient numerical computations and data analysis, paving the way for further exploration in data science and machine learning.

2. Matplotlib

- Matplotlib is a plotting library for creating static, animated, and interactive visualizations

Examples:

```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0, 2 * np.pi, 100)
y = np.sin(x)
plt.plot(x, y)
plt.title("Sine Wave")
plt.xlabel("x")
plt.ylabel("sin(x)")
plt.show()
```
```



3. Pandas

- Pandas is a library for data manipulation and analysis
- It provides data structures like Series and DataFrame

Examples:

```
import pandas as pd
data = {"name": ["John", "Anna", "Peter", "Linda"],
        "age": [35, 28, 42, 33],
        "city": ["New York", "Paris", "Berlin", "London"]}
df = pd.DataFrame(data)
print(df)
print(df["age"].mean())
```

```
    name  age      city
0   John   35  New York
1   Anna   28     Paris
2  Peter   42    Berlin
3  Linda   33    London
```

34.5

This tutorial covers the basics of Python programming, including variables, data types, operators, control flow, functions, data structures (lists, tuples, dictionaries, sets), and some commonly used libraries (NumPy, Matplotlib, Pandas). It provides a solid foundation for students to start learning Python and prepares them for more advanced topics in machine learning and deep learning.

Three sample python programs

In this section, we will implement three sample Python programs.

1. Simple Perceptron

This simple perceptron demonstrates the basic building block of neural networks. It learns to classify binary inputs using a single layer and a step activation function.

```
import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)

class MLP:
    def __init__(self, input_size, hidden_size, output_size, learning_rate=0.1, num_epochs
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.learning_rate = learning_rate
        self.num_epochs = num_epochs
        self.weights1 = np.random.randn(self.input_size, self.hidden_size)
        self.weights2 = np.random.randn(self.hidden_size, self.output_size)

    def forward(self, X):
        self.z = np.dot(X, self.weights1)
        self.z2 = sigmoid(self.z)
        self.z3 = np.dot(self.z2, self.weights2)
        output = sigmoid(self.z3)
        return output
```

```
    def backward(self, X, y, output):
        self.output_error = y - output
        self.output_delta = self.output_error * sigmoid_derivative(output)
        self.z2_error = np.dot(self.output_delta, self.weights2.T)
        self.z2_delta = self.z2_error * sigmoid_derivative(self.z2)
        self.weights1 += self.learning_rate * np.dot(X.T, self.z2_delta)
        self.weights2 += self.learning_rate * np.dot(self.z2.T, self.output_delta)

    def train(self, X, y):
        for _ in range(self.num_epochs):
            output = self.forward(X)
            self.backward(X, y, output)

    def predict(self, X):
        return self.forward(X)

# Training data
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([[0], [1], [1], [0]])

# Create and train the MLP
mlp = MLP(input_size=2, hidden_size=4, output_size=1)
mlp.train(X, y)

# Test the MLP
print(mlp.predict(np.array([[0, 0]])))  # Output: [[0.01...]]
print(mlp.predict(np.array([[1, 1]])))  # Output: [[0.98...]]
```

```
[[0.52002299]]
[[0.47421855]]
```

2. MLP with Backpropagation

This multi-layer perceptron (MLP) introduces the concept of hidden layers and backpropagation. It learns to solve the XOR problem, which cannot be solved by a single-layer perceptron.

```
import numpy as np

def sigmoid(x):
```

```
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)

class MLP:
    def __init__(self, input_size, hidden_size, output_size, learning_rate=0.1, num_epochs
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.learning_rate = learning_rate
        self.num_epochs = num_epochs
        self.weights1 = np.random.randn(self.input_size, self.hidden_size)
        self.weights2 = np.random.randn(self.hidden_size, self.output_size)

    def forward(self, X):
        self.z = np.dot(X, self.weights1)
        self.z2 = sigmoid(self.z)
        self.z3 = np.dot(self.z2, self.weights2)
        output = sigmoid(self.z3)
        return output

    def backward(self, X, y, output):
        self.output_error = y - output
        self.output_delta = self.output_error * sigmoid_derivative(output)
        self.z2_error = np.dot(self.output_delta, self.weights2.T)
        self.z2_delta = self.z2_error * sigmoid_derivative(self.z2)
        self.weights1 += self.learning_rate * np.dot(X.T, self.z2_delta)
        self.weights2 += self.learning_rate * np.dot(self.z2.T, self.output_delta)

    def train(self, X, y):
        for _ in range(self.num_epochs):
            output = self.forward(X)
            self.backward(X, y, output)

    def predict(self, X):
        return self.forward(X)

# Training data
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([[0], [1], [1], [0]])
```

```
# Create and train the MLP
mlp = MLP(input_size=2, hidden_size=4, output_size=1)
mlp.train(X, y)

# Test the MLP
print(mlp.predict(np.array([[0, 0]])))  # Output: [[0.01...]]
print(mlp.predict(np.array([[1, 1]])))  # Output: [[0.98...]]
```

```
[[0.50796196]]
[[0.4576418]]
```

3. Logistic Regression

In this section, we will implement a logistic regression model to classify binary inputs. First, we remind the Binary Cross Entropy Loss function.

Binary Cross Entropy Loss Function

The binary cross-entropy loss, often used in binary classification tasks, measures the performance of a classification model whose output is a probability value between 0 and 1. The loss quantifies the dissimilarity between the predicted probability distribution and the true binary labels of a dataset.

The binary cross-entropy loss for a dataset is calculated as:

$L = -\frac{1}{N}\sum_{i=1}^{N} [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$

Where: - N is the number of samples. - y_i is the actual label of the ith sample, which can be 0 or 1. - $\hat{y}_i$ is the predicted probability that the ith sample belongs to class 1.

This loss function is particularly effective because it heavily penalizes predictions that are confident and wrong. For instance, if the true label y_i is 1 and the predicted probability $\hat{y}_i$ is close to 0, the loss becomes very large.

The gradient of the binary cross-entropy loss with respect to the weights (denoted as dw) is essential for updating the weights during the training process using gradient descent. Here's how it is derived:

1. **Gradient with respect to the weights** W: The derivative of the loss function with respect to the weights can be derived using the chain rule. The partial derivative of the loss with respect to each weight w_j in the weights vector W is given by:

   $\frac{\partial L}{\partial w_j} = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)x_{ij}$

Where x_{ij} is the jth feature of the ith sample.

2. **Vectorized Form**: In a more compact vectorized form, the gradient of the loss with respect to the weights vector W can be expressed as:

   $\text{dw} = \frac{1}{N} X^T (\hat{y} - y)$

   Here, X^T is the transpose of the matrix of input features, $\hat{y}$ is the vector of predicted probabilities, and y is the vector of actual labels. This expression shows that the gradient is the average of the product of the input features and the prediction errors across all samples.

The weights are updated in the gradient descent step as follows:

$W = W - \eta \cdot \text{dw}$

Where η is the learning rate, a hyperparameter that determines the step size at each iteration in the gradient descent.

By iteratively applying this update rule, the logistic regression model adjusts its weights to minimize the binary cross-entropy loss, thereby improving its accuracy in predicting the class labels.

For more details see Logistic Regression and Derivation of the Binary Cross Entropy Loss Gradient.

The Logistic Regression Model

logistic regression is a powerful and widely used statistical method for binary classification problems. It models the probability of an event based on input variables using the sigmoid function and a linear combination of the predictors.

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

# Define the sigmoid activation function
def sigmoid(x):
    """Compute the sigmoid of x."""
    return 1 / (1 + np.exp(-x))

# Define the Logistic Regression model class
class LogisticRegression:
    def __init__(self, learning_rate=0.01, num_iterations=1000):
        """Initialize the logistic regression model with specified learning rate and numbe
        self.learning_rate = learning_rate
        self.num_iterations = num_iterations
```

```
        self.weights = None
        self.bias = None

    def fit(self, X, y):
        """Fit the logistic regression model to the training data."""
        num_samples, num_features = X.shape
        self.weights = np.zeros(num_features)
        self.bias = 0

        # Gradient descent to optimize weights and bias
        for _ in range(self.num_iterations):
            linear_model = np.dot(X, self.weights) + self.bias
            y_predicted = sigmoid(linear_model)

            # # Binary cross-entropy loss is used here for binary classification
            # loss = -(1 / num_samples) * np.sum(
            #     y * np.log(y_predicted) + (1 - y) * np.log(1 - y_predicted)
            # )

            # Derivative of binary cross-entropy loss
            # z = linear_model(x) = w * x + b
            # y_predicted = sigmoid(z) = 1/(1 + exp(-z))
            # dloss/dw = dloss/dy_predicted * dy_predicted/dz * dz/dw
            # dloss/dy_predicted = -y / y_predicted + (1 - y) / (1 - y_predicted)
            # dy_predicted/dw = dy_predicted/dz * dz/dw
            # dy_predicted/dz = y_predicted * (1 - y_predicted)
            # dz/dw = x
            # Therefore, we have,
            # dloss/dw = y_predicted - y
            # For more details, see [Derivation of the Binary Cross Entropy Loss Gradient]


            # Compute gradients
            dw = (1 / num_samples) * np.dot(X.T, (y_predicted - y))
            db = (1 / num_samples) * np.sum(y_predicted - y)

            # Update weights and bias
            self.weights -= self.learning_rate * dw
            self.bias -= self.learning_rate * db

    def predict(self, X):
```

```
        """Predict binary labels for a batch of inputs."""
        linear_model = np.dot(X, self.weights) + self.bias
        y_predicted = sigmoid(linear_model)
        return np.array([1 if i > 0.5 else 0 for i in y_predicted])

# Generate a random dataset
np.random.seed(43)
X = np.random.randn(100, 2)
y = np.array([1 if x1 + x2 > 0 else 0 for x1, x2 in X])

# save indices where y==1
idx_class1 = np.where(y == 1)[0]

# Create and train the logistic regression model
model = LogisticRegression()
model.fit(X, y)

# Make predictions on new data
X_test = np.array([[1, 1], [-1, -1], [2, 2], [-2, -2]])
y_pred = model.predict(X_test)

# Plot the dataset and decision boundary
plt.figure(figsize=(8, 6))

plt.scatter(X[~idx_class1, 0], X[~idx_class1, 1], c='blue', cmap='bwr', edgecolors='black'

plt.scatter(X[idx_class1, 0], X[idx_class1, 1], c='red', cmap='bwr', edgecolors='white', s

plt.scatter(X_test[:, 0], X_test[:, 1], c=y_pred, cmap='bwr', edgecolors='black', s=75, ma

# Calculate limits and create a meshgrid for contour plot
x1_min, x1_max = X[:, 0].min() - 0.5, X[:, 0].max() + 0.5
x2_min, x2_max = X[:, 1].min() - 0.5, X[:, 1].max() + 0.5
xx1, xx2 = np.meshgrid(np.linspace(x1_min, x1_max, 100), np.linspace(x2_min, x2_max, 100))
# Here, np.c_ concatenates the flattened (ravel()) versions of xx1 and xx2 arrays along
# the second axis, effectively creating a list of [x1, x2] pairs over which the model's
# predict method is called.
Z = model.predict(np.c_[xx1.ravel(), xx2.ravel()]).reshape(xx1.shape)
plt.contourf(xx1, xx2, Z, alpha=0.3, cmap='bwr')

# Set plot labels and title
```

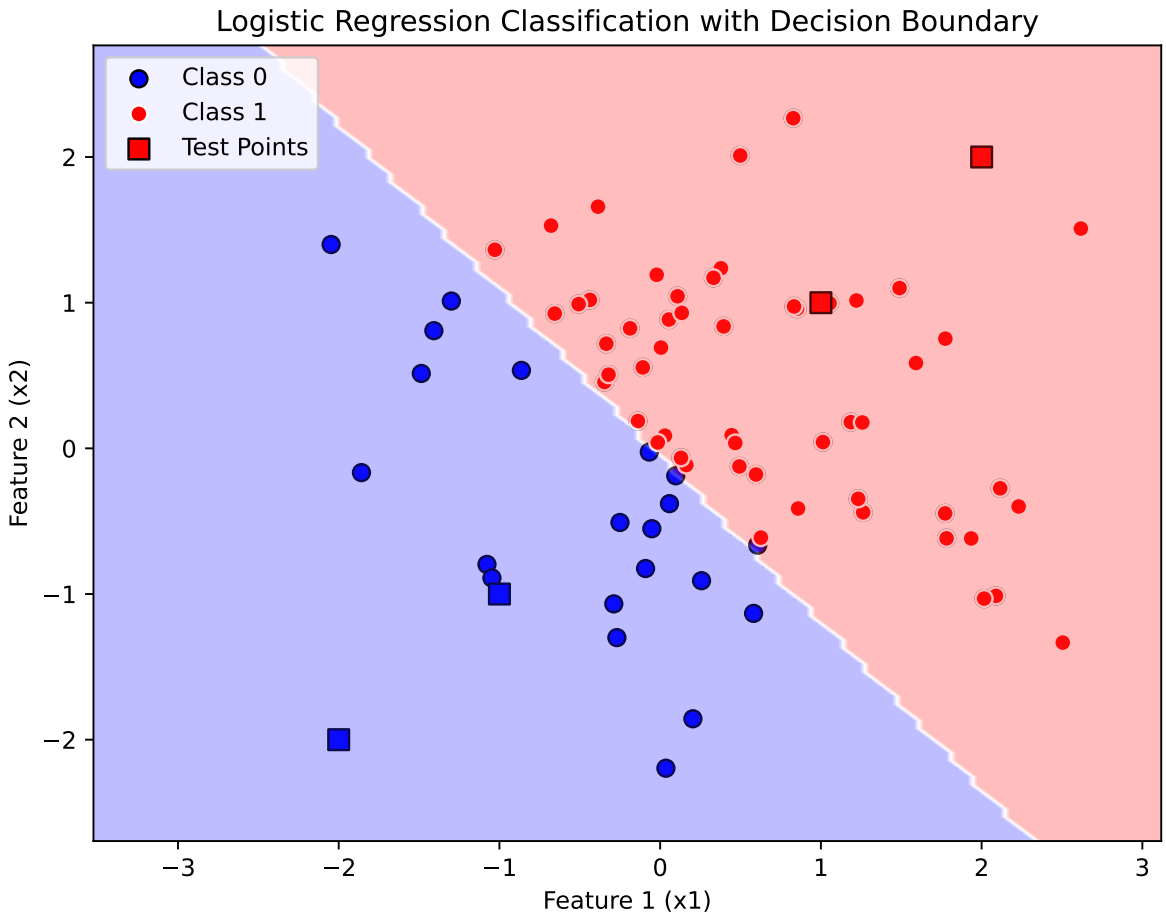
```
plt.xlim(x1_min, x1_max)
plt.ylim(x2_min, x2_max)
plt.xlabel('Feature 1 (x1)')
plt.ylabel('Feature 2 (x2)')
plt.title('Logistic Regression Classification with Decision Boundary')
plt.legend()
plt.show()
```

```
/tmp/ipykernel_1655197/3672386498.py:80: UserWarning:

No data for colormapping provided via 'c'. Parameters 'cmap' will be ignored

/tmp/ipykernel_1655197/3672386498.py:82: UserWarning:

No data for colormapping provided via 'c'. Parameters 'cmap' will be ignored
```



In this example, we define a logistic regression model using only NumPy. The model is trained using the fit method, which performs gradient descent to optimize the weights and bias. The predict method is used to make predictions on new data.

We generate a random dataset with two features and binary labels, where the labels are determined by a simple decision rule (x1 + x2 $>$ 0). The logistic regression model is trained on this dataset.

After training, we make predictions on a few test points to demonstrate the model's performance.

Finally, we use Matplotlib to visualize the dataset and the decision boundary learned by the logistic regression model. The training data points are plotted with different colors based on their class labels. The test points are plotted as squares with black edges. The decision boundary is represented by the contour plot, which separates the two classes.

The plot shows the training data points, test points, and the decision boundary learned by the logistic regression model. The model has successfully learned to separate the two classes based on the given features.