Ex.1 Revise NumPy

```
import numpy as np
```

```
# Shape and type of the array.

a = np.array([0,2,4])
print(a)
print("Shape of a:\t",a.shape)
print("Datatype of a:\t",a.dtype)
```

```
[0 2 4]
Shape of a:      (3,)
Datatype of a:   int64
```

```
# Access specific elements of a 2D NumPy array.

b = np.array([(0,2,4),(0,1,7)])
print("The array:")
print(b)
print("\nThe first row:",b[0])
print("\nThe first row second element:",b[0][1])
```

```
The array:
[[0 2 4]
 [0 1 7]]

The first row: [0 2 4]

The first row second element: 2
```

```
# Show how to slice a NumPy array to get all rows, but only the first two columns.

print("The first 2 coloumns of all rows:\n",b[0:2,:2])
```

```
The first 2 coloumns of all rows:
 [[0 2]
 [0 1]]
```

```
# Reshape a 1D array of size 12 into a 2D array with 3 rows and 4 columns.

c = np.random.randint(0,10,size = 12)
print("The 1D array:\n",c)
d = c.reshape(3,4)
print("\nThe 2D array:\n",d)
```

```
The 1D array:
 [4 5 6 7 9 3 4 6 2 8 3 9]

The 2D array:
```

```
[[4 5 6 7]
 [9 3 4 6]
 [2 8 3 9]]
```

```python
# Perform matrix multiplication between two 2D arrays using NumPy.

A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
result = np.dot(A, B)
print("Matrix Multiplication:\n", result)
```

```
Matrix Multiplication:
 [[19 22]
 [43 50]]
```

```python
# Compute the mean, median, and standard deviation of a NumPy array.

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])
mean = np.mean(arr)
median = np.median(arr)
std_dev = np.std(arr)
print(f"Mean: {mean}, Median: {median}, Standard Deviation: {std_dev}")
```

```
Mean: 5.0, Median: 5.0, Standard Deviation: 2.581988897471611
```

```python
# Perform vertical and horizontal stacking in NumPy.

arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

vertical_stack = np.vstack((arr1, arr2))
print("Vertical Stacking:\n", vertical_stack)

horizontal_stack = np.hstack((arr1, arr2))
print("\nHorizontal Stacking:\n", horizontal_stack)
```

```
Vertical Stacking:
 [[1 2 3]
 [4 5 6]]

Horizontal Stacking:
 [1 2 3 4 5 6]
```

```python
# Flatten a 3 x 4 NumPy array into a 1D array.

arr = np.array([[1, 2, 3, 4],
                [5, 6, 7, 8],
                [9, 10, 11, 12]])
flattened_arr = arr.flatten()
print("Flattened Array:", flattened_arr)
```

```
Flattened Array: [ 1  2  3  4  5  6  7  8  9 10 11 12]
```

```
# Generate a random array of size n with values drawn from a normal distribution.

n = 5
random_arr = np.random.normal(size=n)
print("Random Array from Normal Distribution:", random_arr)
```

```
Random Array from Normal Distribution: [ 0.45537544 -0.09346873 -0.22733775  0.079052
```

```
# Perform element-wise addition, subtraction, multiplication, and division.

arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

addition = arr1 + arr2
subtraction = arr1 - arr2
multiplication = arr1 * arr2
division = arr1 / arr2

print("Element-wise Addition:\t\t", addition)
print("Element-wise Subtraction:\t", subtraction)
print("Element-wise Multiplication:\t", multiplication)
print("Element-wise Division:\t\t", division)
```

```
Element-wise Addition:          [5 7 9]
Element-wise Subtraction:       [-3 -3 -3]
Element-wise Multiplication:    [ 4 10 18]
Element-wise Division:          [0.25 0.4  0.5 ]
```

Ex.2 Practice OpenCV

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Load the image using OpenCV
image = cv2.imread('cameraman.jpg')

# Convert the image from BGR to RGB as OpenCV uses BGR by default
image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

# Display the image using Matplotlib
plt.imshow(image_rgb)
plt.axis('off')
plt.show()
```

```
## Resize the image to half of its original size using OpenCV

# The original dimensions of the image
original_height, original_width = image.shape[:2]

# Resize the image to half of its original size
resized_image = cv2.resize(image, (original_width // 2, original_height // 2))

plt.imshow(cv2.cvtColor(resized_image, cv2.COLOR_BGR2RGB))
plt.axis('off')
plt.show()
```

```
## Crop a specific region (top-left quarter) of an image using NumPy slicing in OpenCV

cropped_image = image[:original_height // 2, :original_width // 2]
plt.imshow(cv2.cvtColor(cropped_image, cv2.COLOR_BGR2RGB))
plt.axis('off')
plt.show()
```

```
## Convert the image from BGR to Grayscale

gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
```

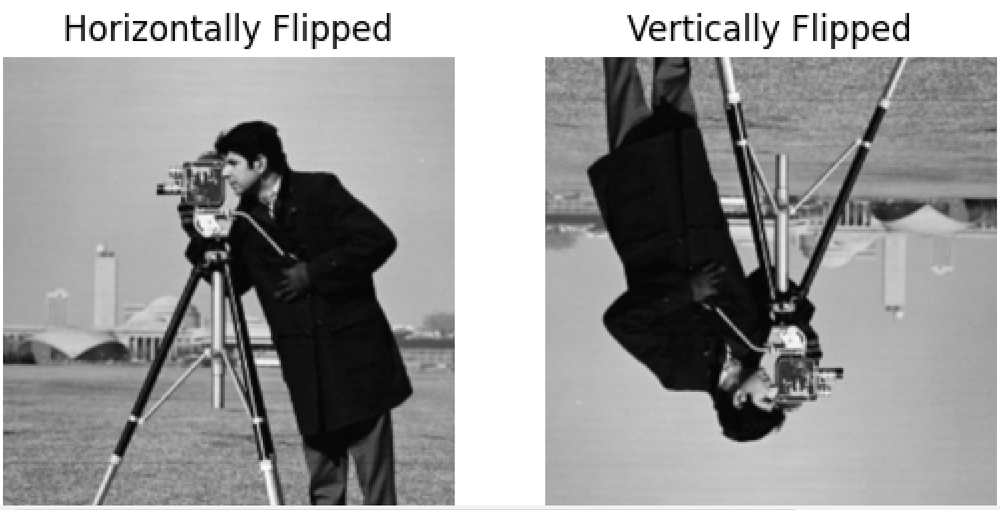
```
plt.imshow(gray_image, cmap='gray')
plt.axis('off')
plt.show()
```

```
## Display the image using Matplotlib instead of OpenCV
plt.imshow(image_rgb)
plt.axis('off')
plt.show()
```

```
## Find the dimensions (width, height, and channels) of an image using OpenCV

height, width, channels = image.shape
print(f"Width: {width}, Height: {height}, Channels: {channels}")
```

```
Width: 320, Height: 320, Channels: 3
```

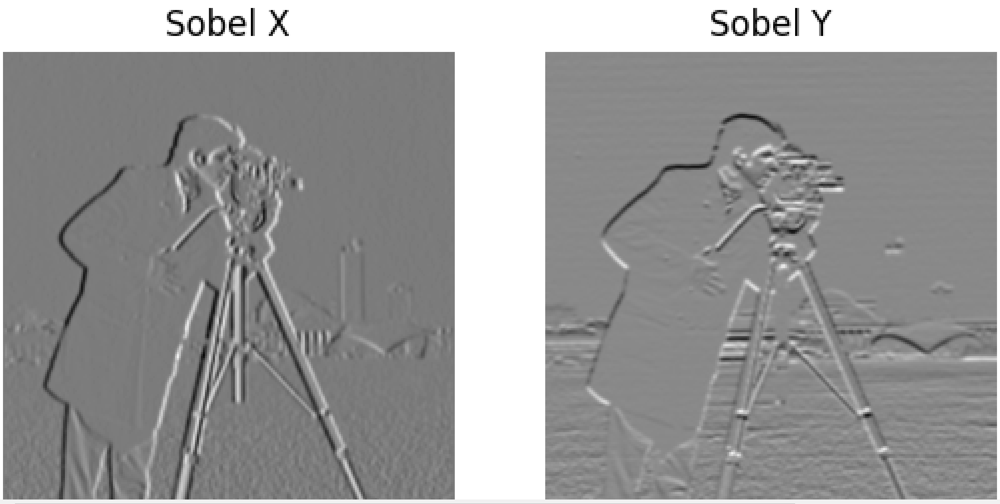
```
## Flip the image horizontally and vertically using OpenCV

horizontally_flipped = cv2.flip(image, 1)
vertically_flipped = cv2.flip(image, 0)

plt.subplot(1, 2, 1)
plt.imshow(cv2.cvtColor(horizontally_flipped, cv2.COLOR_BGR2RGB))
plt.axis('off')
plt.title("Horizontally Flipped")

plt.subplot(1, 2, 2)
plt.imshow(cv2.cvtColor(vertically_flipped, cv2.COLOR_BGR2RGB))
plt.axis('off')
plt.title("Vertically Flipped")
plt.show()
```




```
## Apply a 5x5 averaging (box) filter to smoothen the image

smoothed_image = cv2.blur(image, (5, 5))
plt.imshow(cv2.cvtColor(smoothed_image, cv2.COLOR_BGR2RGB))
plt.axis('off')
plt.show()
```

```
## Apply a sharpening kernel to enhance the edges and details in the image

sharpening_kernel = np.array([[-1, -1, -1],
                              [-1, 9, -1],
                              [-1, -1, -1]])

sharpened_image = cv2.filter2D(image, -1, sharpening_kernel)

plt.imshow(cv2.cvtColor(sharpened_image, cv2.COLOR_BGR2RGB))
plt.axis('off')
plt.show()
```

```
## Apply the Sobel operator to detect edges in both the horizontal and vertical direction

# Apply the Sobel operator to detect edges in the horizontal direction
sobelx = cv2.Sobel(gray_image, cv2.CV_64F, 1, 0, ksize=5)

# Apply the Sobel operator to detect edges in the vertical direction
sobely = cv2.Sobel(gray_image, cv2.CV_64F, 0, 1, ksize=5)

plt.subplot(1, 2, 1)
plt.imshow(sobelx, cmap='gray')
plt.axis('off')
plt.title("Sobel X")

plt.subplot(1, 2, 2)
plt.imshow(sobely, cmap='gray')
plt.axis('off')
plt.title("Sobel Y")
plt.show()
```



Ex.3 Practice PyTorch

```
## Create two random 3x3 tensors and perform matrix multiplication.
## Compute the matrix product and use autograd to calculate the gradient.

import torch

A = torch.rand(3, 3, requires_grad=True)
B = torch.rand(3, 3)

C = torch.matmul(A, B)

C.sum().backward()

print("Gradient of C with respect to A:\n", A.grad)
```

```
Gradient of C with respect to A:
 tensor([[1.5237, 1.7642, 1.4962],
        [1.5237, 1.7642, 1.4962],
        [1.5237, 1.7642, 1.4962]])
```

```
## Perform element-wise operations on tensors with broadcasting.

# Create tensors for broadcasting
A = torch.rand(3, 1)  # 3x1 tensor
B = torch.rand(1, 3)  # 1x3 tensor
C = torch.rand(3, 3)  # 3x3 tensor

# Broadcasting addition and multiplication
result = (A + B) * C
print("Result of broadcasting addition and element-wise multiplication:\n", result)
```

```
Result of broadcasting addition and element-wise multiplication:
 tensor([[1.6721, 0.3958, 0.8421],
        [0.5853, 0.0197, 0.4920],
        [0.1101, 0.2051, 0.3348]])
```

```
## Reshape a 2D tensor and extract specific slices

# Create a 2D tensor of shape (6, 4)
tensor = torch.rand(6, 4)

# Reshape it into a tensor of shape (3, 8)
reshaped_tensor = tensor.view(3, 8)

# Extract slices: all rows, first two columns
sliced_tensor = reshaped_tensor[:, :2]

print("Original Tensor:\n", tensor)
print("\nReshaped Tensor:\n", reshaped_tensor)
print("\nSliced Tensor (all rows, first two columns):\n", sliced_tensor)
```

```
Original Tensor:
 tensor([[0.4024, 0.8094, 0.0679, 0.7987],
        [0.1331, 0.5087, 0.8362, 0.5025],
        [0.3847, 0.6609, 0.2165, 0.1363],
        [0.5296, 0.8556, 0.9632, 0.3239],
        [0.2094, 0.9439, 0.7092, 0.1273],
        [0.0818, 0.3460, 0.2259, 0.9417]])

Reshaped Tensor:
 tensor([[0.4024, 0.8094, 0.0679, 0.7987, 0.1331, 0.5087, 0.8362, 0.5025],
        [0.3847, 0.6609, 0.2165, 0.1363, 0.5296, 0.8556, 0.9632, 0.3239],
        [0.2094, 0.9439, 0.7092, 0.1273, 0.0818, 0.3460, 0.2259, 0.9417]])

Sliced Tensor (all rows, first two columns):
 tensor([[0.4024, 0.8094],
        [0.3847, 0.6609],
        [0.2094, 0.9439]])
```

```
## Convert a NumPy array into a PyTorch tensor, perform operations, and convert it back t

import numpy as np

np_array = np.array([[1, 2, 3], [4, 5, 6]])

# Convert NumPy array to PyTorch tensor
torch_tensor = torch.from_numpy(np_array).float()

# Performing Multiplication
modified_tensor = torch_tensor * 2

# Convert the result back to a NumPy array
modified_np_array = modified_tensor.numpy()

print("Original NumPy Array:\n", np_array)
print("\nModified PyTorch Tensor:\n", modified_tensor)
print("\nModified NumPy Array:\n", modified_np_array)
```

```
Original NumPy Array:
 [[1 2 3]
 [4 5 6]]
```

```
Modified PyTorch Tensor:
 tensor([[ 2.,  4.,  6.],
        [ 8., 10., 12.]])

Modified NumPy Array:
 [[ 2.  4.  6.]
 [ 8. 10. 12.]]
```

```
## Initialize 5x5 tensors from uniform and normal distributions, perform elementwise mult

# Initialize a 5x5 tensor with values from a uniform distribution between 0 and 1
uniform_tensor = torch.rand(5, 5)

# Initialize a 5x5 tensor with values from a normal distribution (mean 0, std 1)
normal_tensor = torch.randn(5, 5)

# Perform element-wise multiplication
result_tensor = uniform_tensor * normal_tensor

# Compute the mean and standard deviation of the resulting tensor
mean_value = result_tensor.mean()
std_value = result_tensor.std()

# Reshape the result into a 1D tensor of size 25
reshaped_tensor = result_tensor.view(25)

# Compute the sum of all elements in the reshaped tensor
sum_value = reshaped_tensor.sum()

print("Mean of result tensor:\t\t\t", mean_value.item())
print("Standard deviation of result tensor:\t", std_value.item())
print("Sum of all elements:\t\t\t", sum_value.item())
```

```
Mean of result tensor:                  0.0989852175116539
Standard deviation of result tensor:    0.5546156167984009
Sum of all elements:                    2.474630355834961
```

Ex.4

```
## Build a function that returns the sigmoid of a real number using math.exp() and np.exp

import math
import numpy as np

# Sigmoid function using math.exp
def sigmoid_math(x):
    return 1 / (1 + math.exp(-x))

# Sigmoid function using np.exp
def sigmoid_numpy(x):
    return 1 / (1 + np.exp(-x))
```

```
x = 0.5
print("Sigmoid using math.exp:", sigmoid_math(x))
print("Sigmoid using np.exp:", sigmoid_numpy(x))

# np.exp() is preferable because it can handle arrays and operates element-wise.
# math.exp() only works for scalars, so it's less versatile compared to np.exp() when dea
arr = np.array([0.5, 1.0, 1.5])
print("Sigmoid of array using np.exp:", sigmoid_numpy(arr))
```

```
Sigmoid using math.exp: 0.6224593312018546
Sigmoid using np.exp: 0.6224593312018546
Sigmoid of array using np.exp: [0.62245933 0.73105858 0.81757448]
```

```
## Implement the gradient of the sigmoid function (sigmoid_grad)

# Gradient of sigmoid function
def sigmoid_grad(x):
    sig = sigmoid_numpy(x)  # Reusing the sigmoid function
    return sig * (1 - sig)

x = np.array([0.5, 1.0, 1.5])
print("Gradient of sigmoid:", sigmoid_grad(x))
```

```
Gradient of sigmoid: [0.23500371 0.19661193 0.14914645]
```

```
## Implement image2vector() to convert an image of shape (length, height, 3) into a vecto

def image2vector(image):
    return image.reshape(-1, 1)

image = np.random.rand(3, 3, 3)  # 3x3 image with 3 color channels
vector = image2vector(image)
print("Image shape:", image.shape)
print("Vector shape:", vector.shape)
```

```
Image shape: (3, 3, 3)
Vector shape: (27, 1)
```

```
## Implement normalizeRows() to normalize the rows of a matrix.
## After applying this function to an input matrix x, each row of x should be a vector of

def normalizeRows(x):
    row_norms = np.linalg.norm(x, axis=1, keepdims=True)
    normalized_x = x / row_norms
    return normalized_x

x = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])

normalized_x = normalizeRows(x)
```

```
row_lengths = np.linalg.norm(normalized_x, axis=1)

print("Original matrix:\n",x)
print("\nNormalized matrix:\n", normalized_x)
print("\nL2 norm of each row (should be 1):\n", row_lengths)
```

```
Original matrix:
 [[1 2 3]
 [4 5 6]
 [7 8 9]]

Normalized matrix:
 [[0.26726124 0.53452248 0.80178373]
 [0.45584231 0.56980288 0.68376346]
 [0.50257071 0.57436653 0.64616234]]

L2 norm of each row (should be 1):
 [1. 1. 1.]
```

```
## Implement loss functions L1 and L2

# L1 loss function
def l1_loss(y_true, y_pred):
    return np.mean(np.abs(y_true - y_pred))

# L2 loss function
def l2_loss(y_true, y_pred):
    return np.mean(np.square(y_true - y_pred))

# Example usage
y_true = np.array([3.0, -0.5, 2.0, 7.0])
y_pred = np.array([2.5, 0.0, 2.1, 7.8])

# Compute L1 and L2 loss
l1 = l1_loss(y_true, y_pred)
l2 = l2_loss(y_true, y_pred)

print(f"L1 Loss (Mean Absolute Error):\t {l1}")
print(f"L2 Loss (Mean Squared Error) :\t {l2}")
```

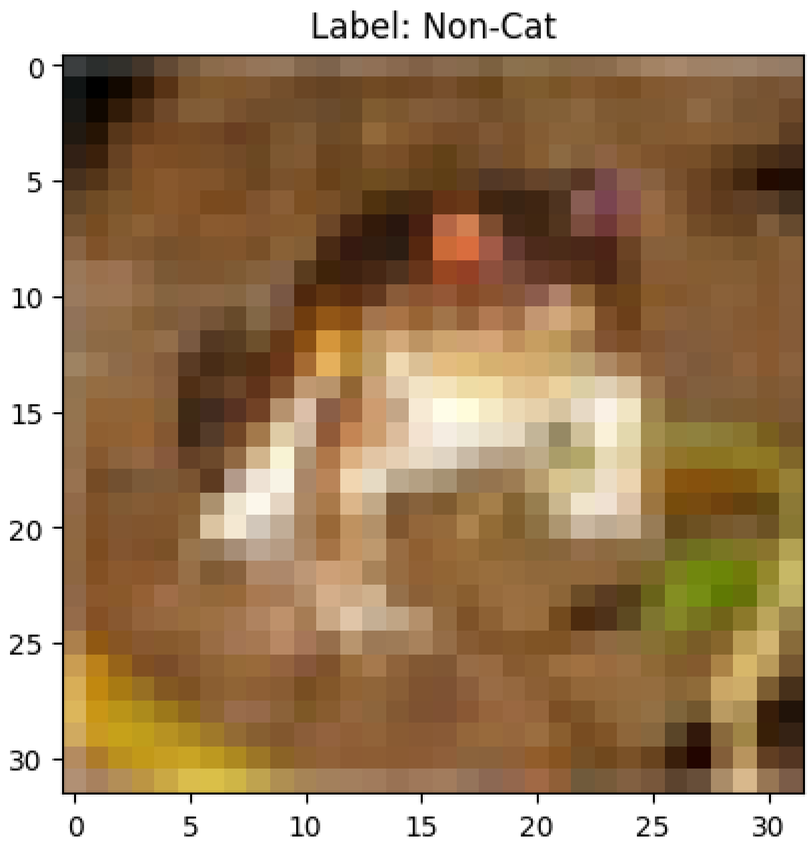
```
L1 Loss (Mean Absolute Error):   0.475
L2 Loss (Mean Squared Error) :   0.2874999999999999
```

Ex.5 Towards neural network from logistic regression

```
## Download and Load the Dataset

import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import cifar10

# Load the CIFAR-10 dataset
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
```

```
# Get only cat (label = 3) and non-cat (label != 3)
cat_label = 3
y_train_cat = (y_train == cat_label).astype(int)  # 1 for cat, 0 for non-cat
y_test_cat = (y_test == cat_label).astype(int)
```

```
## Load and Display the First Image

# Display the first image in the training dataset
first_image = X_train[0]
plt.imshow(first_image)
plt.title(f"Label: {'Cat' if y_train_cat[0] else 'Non-Cat'}")
plt.show()

# Print the shape of the first image
print("Shape of the first image:", first_image.shape)  # Should be (32, 32, 3) for CIFAR-
```




```
##  Implement Logistic Regression for Image Classification

# Reshape (flatten) the training and test images
X_train_flatten = X_train.reshape(X_train.shape[0], -1)  # Shape (50000, 32*32*3)
X_test_flatten = X_test.reshape(X_test.shape[0], -1)     # Shape (10000, 32*32*3)

# Normalize the data (optional but recommended for gradient-based models)
X_train_flatten = X_train_flatten / 255.0
X_test_flatten = X_test_flatten / 255.0
```

```
from sklearn.linear_model import LogisticRegression

# Initialize the logistic regression model
log_reg_model = LogisticRegression(max_iter=1000)

# Train the model on the training data
log_reg_model.fit(X_train_flatten, y_train_cat.ravel())

# Predict on the test set
y_pred = log_reg_model.predict(X_test_flatten)
```

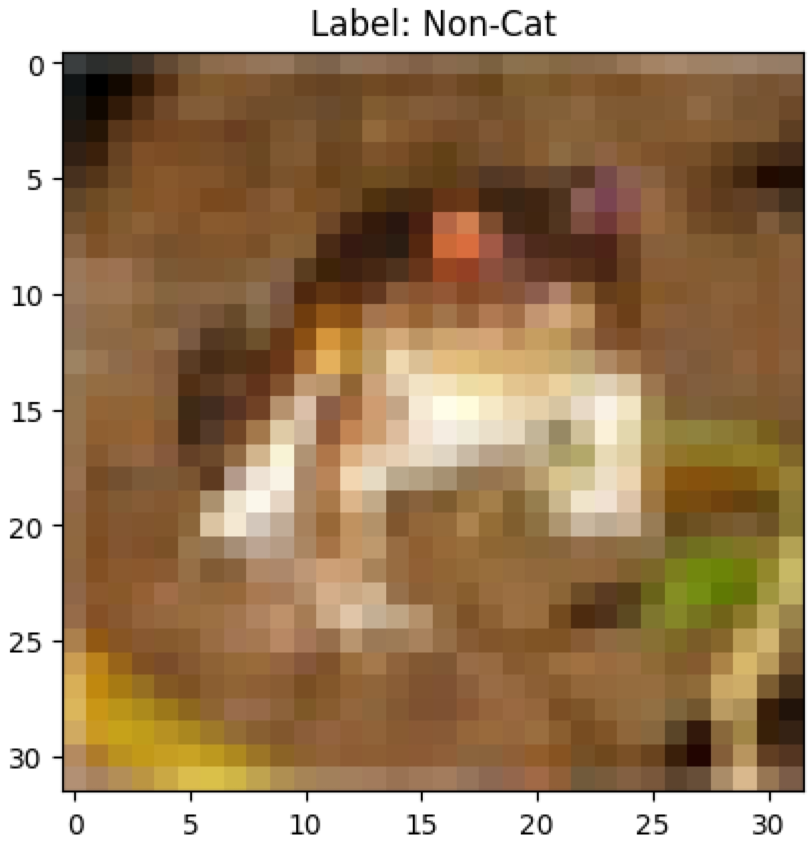
```
/usr/local/lib/python3.10/dist-packages/sklearn/linear_model/_logistic.py:460: Conver
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
```

```
from sklearn.metrics import accuracy_score, classification_report

# Calculate the accuracy of the model
accuracy = accuracy_score(y_test_cat, y_pred)
print(f"Test Accuracy: {accuracy * 100:.2f}%")

# Print classification report for more details
print(classification_report(y_test_cat, y_pred, target_names=["Non-Cat", "Cat"]))
```

```
Test Accuracy: 89.60%
              precision    recall  f1-score   support

     Non-Cat       0.90      0.99      0.94      9000
         Cat       0.33      0.04      0.07      1000

    accuracy                           0.90     10000
   macro avg       0.62      0.52      0.51     10000
weighted avg       0.85      0.90      0.86     10000
```

```
import numpy as np
from tensorflow.keras.datasets import cifar10
import matplotlib.pyplot as plt


# 5.1 Download dataset
(X_train_full, y_train_full), (X_test_full, y_test_full) = cifar10.load_data()

# 5.2 Load and display the first image from the training dataset
# Display the first image in the training set
plt.imshow(X_train_full[0])  # Display the first image from the training set
plt.title(f"Label: {'Cat' if y_train_full[0] == 3 else 'Non-Cat'}")  # Display if it's ca
plt.show()
```

```
# Print the shape of the first image
print("Shape of the first image (RGB):", X_train_full[0].shape)

# We are interested in cats (label=3) and non-cats (all other labels)
# Let's create binary labels: cat (1) and non-cat (0)
y_train_full_binary = np.where(y_train_full == 3, 1, 0)  # Convert labels to 1 (cat) and
y_test_full_binary = np.where(y_test_full == 3, 1, 0)

# Check original shapes
print("Original X_train shape:", X_train_full.shape)
print("Original y_train shape:", y_train_full.shape)

# Randomly shuffle and select 2000 samples for training and 800 for testing
np.random.seed(42)  # Set a seed for reproducibility

train_indices = np.random.choice(X_train_full.shape[0], 2000, replace=False)  # Randomly
test_indices = np.random.choice(X_test_full.shape[0], 800, replace=False)     # Randomly

X_train = X_train_full[train_indices]  # Select the corresponding images
y_train = y_train_full_binary[train_indices]  # Select the corresponding labels

X_test = X_test_full[test_indices]  # Select the corresponding test images
y_test = y_test_full_binary[test_indices]  # Select the corresponding test labels

# Reshape (flatten) the training and test images
X_train_flatten = X_train.reshape(X_train.shape[0], -1)  # Flatten from (32, 32, 3) to (2
X_test_flatten = X_test.reshape(X_test.shape[0], -1)     # Flatten from (32, 32, 3) to (8

# Normalize the data (pixel values are between 0 and 255, so normalize to [0, 1])
X_train_flatten = X_train_flatten / 255.0
X_test_flatten = X_test_flatten / 255.0

# Check the final shapes
print("X_train shape after flattening:", X_train_flatten.shape)
print("y_train shape:", y_train.shape)
print("X_test shape after flattening:", X_test_flatten.shape)
print("y_test shape:", y_test.shape)

# 5.3 Implement Logistic Regression for image classification
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# Initialize and train the logistic regression model
log_reg = LogisticRegression(max_iter=1000)
log_reg.fit(X_train_flatten, y_train.ravel())

# Predict on the test set
y_pred = log_reg.predict(X_test_flatten)

# Compute the accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Test Accuracy: {accuracy * 100:.2f}%")
```




```
Shape of the first image (RGB): (32, 32, 3)
Original X_train shape: (50000, 32, 32, 3)
Original y_train shape: (50000, 1)
X_train shape after flattening: (2000, 3072)
y_train shape: (2000, 1)
X_test shape after flattening: (800, 3072)
y_test shape: (800, 1)
```

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.