Experiment 1

1. Introduction to Basic Python Operations for Data Handling and Analysis

Code Snippet:

# Importing required libraries

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn.linear\_model import LinearRegression

# 1. Pandas - Creating a DataFrame and performing basic operations

print("Pandas Example:")

data = {

'Name': ['Alice', 'Bob', 'Charlie'],

'Age': [25, 30, 35],

'Score': [85, 90, 88]

}

df = pd.DataFrame(data)

print("DataFrame:\n", df)

# Selecting a column

print("\nSelecting 'Name' column:\n", df['Name'])

# Adding a new column

df['Passed'] = df['Score'] > 80

print("\nUpdated DataFrame:\n", df)

# 2. NumPy - Creating arrays and basic operations

print("\nNumPy Example:")

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

print("Array:", array)

# Operations

print("Array + 10:", array + 10)

print("Mean of array:", np.mean(array))

# 3. Plotting - Creating a simple plot

print("\nPlotting Example:")

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

y = [2, 4, 6, 8, 10]

plt.plot(x, y, label="y=2x", color='blue', marker='o')

plt.title("Line Plot Example")

plt.xlabel("X-axis")

plt.ylabel("Y-axis")

plt.legend()

plt.grid(True)

plt.show()

# 4. Create a list and perform basic operations

print("\nList Example:")

my\_list = [10, 20, 30, 40, 50]

print("Original List:", my\_list)

# Add an element

my\_list.append(60)

print("List after adding 60:", my\_list)

# Remove an element

my\_list.remove(20)

print("List after removing 20:", my\_list)

# 5. Scikit-learn (Scilear) - Performing a basic regression

print("\nScikit-learn Example:")

# Short dataset for regression

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

y = np.array([1.5, 3.0, 4.5, 6.0, 7.5])

# Create a Linear Regression model

model = LinearRegression()

model.fit(X, y)

# Predict values

predictions = model.predict(X)

print("Predictions:", predictions)

# Plot regression line

plt.scatter(X, y, color='red', label="Data Points")

plt.plot(X, predictions, color='blue', label="Regression Line")

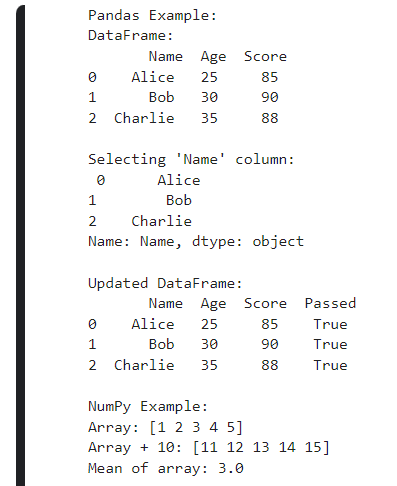
plt.title("Linear Regression Example")

plt.xlabel("X-axis")

plt.ylabel("Y-axis")

plt.legend()

plt.show()

Output

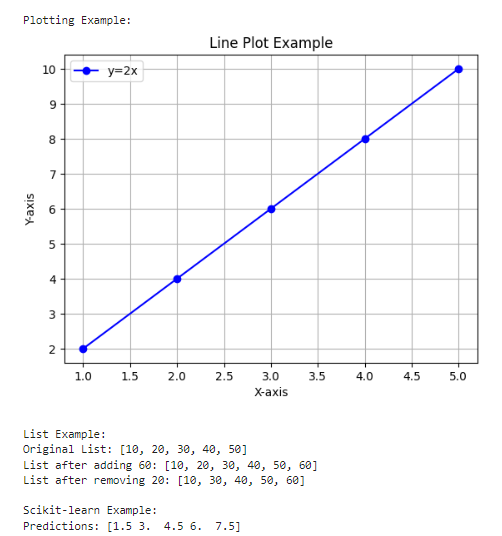
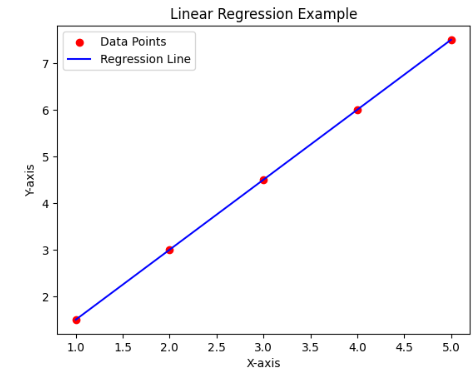


Figure 3.

Figure 2.

Figure 1.

Experiment 2

1. Implementation of a Single-Layer Perceptron in Python

Code Snippet:

import numpy as np

# Define the activation function (step function)

def step\_function(x):

return np.where(x >= 0, 1, 0)

# Define the single-layer perceptron class

class SingleLayerPerceptron:

def \_\_init\_\_(self, input\_size, learning\_rate=0.01):

self.weights = np.random.rand(input\_size) # Initialize weights randomly

self.bias = np.random.rand() # Initialize bias randomly

self.learning\_rate = learning\_rate

def predict(self, inputs):

# Compute the linear combination

linear\_output = np.dot(inputs, self.weights) + self.bias

# Apply the activation function

return step\_function(linear\_output)

def train(self, X, y, epochs=5):

for epoch in range(epochs):

print(f"\nEpoch {epoch + 1}/{epochs}")

for i in range(len(X)):

# Make a prediction

prediction = self.predict(X[i])

# Compute the error

error = y[i] - prediction

# Update weights and bias

self.weights += self.learning\_rate \* error \* X[i]

self.bias += self.learning\_rate \* error

# Print the progress

print(f"Sample {i+1}, Error: {error}, Weights: {self.weights}, Bias: {self.bias}")

# Example dataset (AND logic gate)

X = np.array([

[0, 0],

[0, 1],

[1, 0],

[1, 1]

])

y = np.array([0, 0, 0, 1]) # AND gate output

# Initialize and train the perceptron

perceptron = SingleLayerPerceptron(input\_size=2, learning\_rate=0.1)

perceptron.train(X, y, epochs=10)

# Test the trained perceptron

print("\nTesting the trained perceptron:")

for i in range(len(X)):

print(f"Input: {X[i]}, Predicted: {perceptron.predict(X[i])}, Actual: {y[i]}")

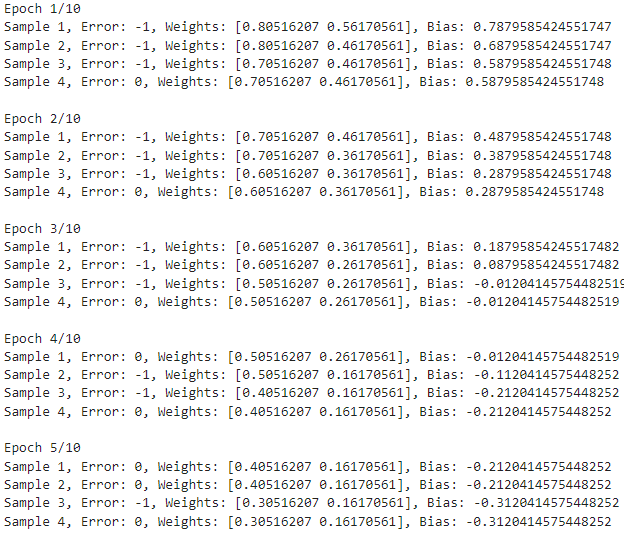
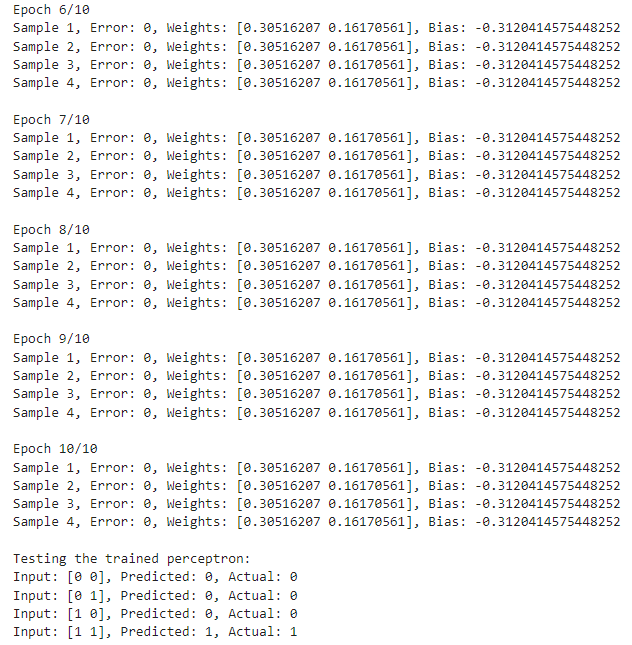
Output

Figure 1

Figure 2

Experiment 3

1. Implementation of ADALINE.

Code Snippet:

import numpy as np

class ADALINE:

def \_\_init\_\_(self, input\_size, learning\_rate=0.01):

"""

Initialize the ADALINE model.

Parameters:

input\_size (int): Number of features in the input data.

learning\_rate (float): Learning rate for weight updates.

"""

self.weights = np.zeros(input\_size)

self.bias = 0.0

self.learning\_rate = learning\_rate

def predict(self, X):

"""

Make predictions using the current weights and bias.

Parameters:

X (array): Input feature array.

Returns:

array: Predicted values (linear output).

"""

return np.dot(X, self.weights) + self.bias

def train(self, X, y, epochs=10):

"""

Train the ADALINE model using the LMS rule.

Parameters:

X (array): Input features (2D array).

y (array): Target values (1D array).

epochs (int): Number of training iterations over the dataset.

"""

for epoch in range(epochs):

print(f"Epoch {epoch + 1}/{epochs}")

# Compute predictions for all samples

predictions = self.predict(X)

# Calculate errors

errors = y - predictions

# Update weights and bias using the LMS rule

self.weights += self.learning\_rate \* np.dot(X.T, errors)

self.bias += self.learning\_rate \* errors.sum()

# Compute mean squared error

mse = (errors\*\*2).mean()

print(f"Mean Squared Error: {mse:.4f}")

print(f"Weights: {self.weights}, Bias: {self.bias}\n")

# Input dataset: AND Gate

X = np.array([

[0, 0], # Input 1

[0, 1], # Input 2

[1, 0], # Input 3

[1, 1] # Input 4

])

y = np.array([0, 0, 0, 1]) # Target output for AND Gate

# Initialize ADALINE model

adaline = ADALINE(input\_size=2, learning\_rate=0.1)

# Train the ADALINE model

adaline.train(X, y, epochs=10)

# Test the trained ADALINE model

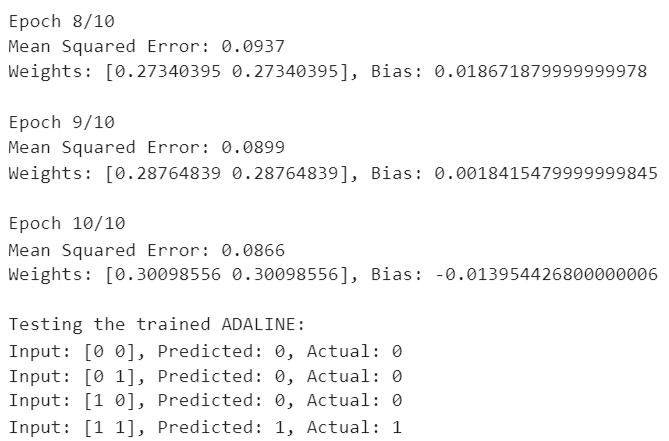
print("Testing the trained ADALINE:")

for sample, label in zip(X, y):

prediction = adaline.predict(sample)

output = 1 if prediction >= 0.5 else 0 # Apply threshold for binary classification

print(f"Input: {sample}, Predicted: {output}, Actual: {label}")

Output

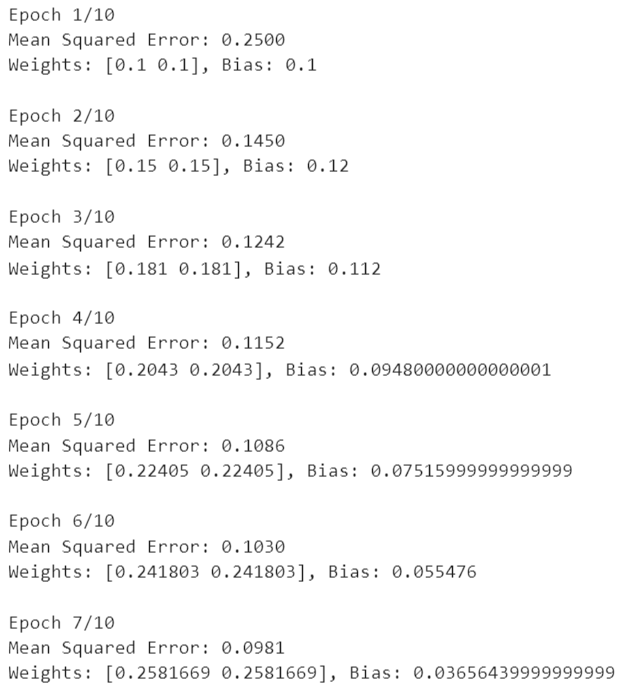


Figure 3

Figure 1

Experiment 4

1. Implementation of Backpropagation Network.

Code Snippet:

import numpy as np

from sklearn.datasets import load\_iris

from sklearn.preprocessing import StandardScaler

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score

class BackpropagationNetwork:

def \_\_init\_\_(self, input\_size, hidden\_size, output\_size, learning\_rate=0.1):

"""Initialize the Backpropagation Network."""

self.hidden\_weights = np.random.randn(input\_size, hidden\_size) \* 0.01

self.hidden\_bias = np.zeros((1, hidden\_size))

self.output\_weights = np.random.randn(hidden\_size, output\_size) \* 0.01

self.output\_bias = np.zeros((1, output\_size))

self.learning\_rate = learning\_rate

def sigmoid(self, x):

"""Sigmoid activation function."""

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

def sigmoid\_derivative(self, x):

"""Derivative of the sigmoid function."""

return x \* (1 - x)

def forward(self, X):

"""Forward propagation."""

self.hidden\_layer\_input = np.dot(X, self.hidden\_weights) + self.hidden\_bias

self.hidden\_layer\_output = self.sigmoid(self.hidden\_layer\_input)

self.output\_layer\_input = np.dot(self.hidden\_layer\_output, self.output\_weights) + self.output\_bias

self.output\_layer\_output = self.sigmoid(self.output\_layer\_input)

return self.output\_layer\_output

def backward(self, X, y, output):

"""Backward propagation to update weights and biases."""

output\_error = y - output

output\_delta = output\_error \* self.sigmoid\_derivative(output)

hidden\_error = np.dot(output\_delta, self.output\_weights.T)

hidden\_delta = hidden\_error \* self.sigmoid\_derivative(self.hidden\_layer\_output)

self.output\_weights += np.dot(self.hidden\_layer\_output.T, output\_delta) \* self.learning\_rate

self.output\_bias += np.sum(output\_delta, axis=0, keepdims=True) \* self.learning\_rate

self.hidden\_weights += np.dot(X.T, hidden\_delta) \* self.learning\_rate

self.hidden\_bias += np.sum(hidden\_delta, axis=0, keepdims=True) \* self.learning\_rate

def train(self, X, y, epochs):

"""Train the Backpropagation Network."""

for epoch in range(epochs):

output = self.forward(X)

self.backward(X, y, output)

loss = np.mean(np.square(y - output))

if (epoch + 1) % 10 == 0: # Print loss every 10 epochs

print(f"Epoch {epoch + 1}/{epochs}, Loss: {loss:.4f}")

def predict(self, X):

"""Make predictions with the trained network."""

return self.forward(X)

# Load the Iris dataset

iris = load\_iris()

X = iris.data

y = iris.target

# One-hot encoding for the target labels

y\_onehot = np.zeros((y.size, y.max() + 1))

y\_onehot[np.arange(y.size), y] = 1

# Standardize the input features

scaler = StandardScaler()

X = scaler.fit\_transform(X)

# Train-test split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y\_onehot, test\_size=0.3, random\_state=42)

# Initialize and train the Backpropagation Network

bpn=BackpropagationNetwork(input\_size=X\_train.shape[1],hidden\_size=5, output\_size=3, learning\_rate=0.1)

bpn.train(X\_train, y\_train, epochs=200)

# Make predictions and evaluate the model

predictions = bpn.predict(X\_test)

predictions = np.argmax(predictions, axis=1)

y\_test\_labels = np.argmax(y\_test, axis=1)

accuracy = accuracy\_score(y\_test\_labels, predictions)

print(f"\nModel Accuracy: {accuracy \* 100:.2f}%")

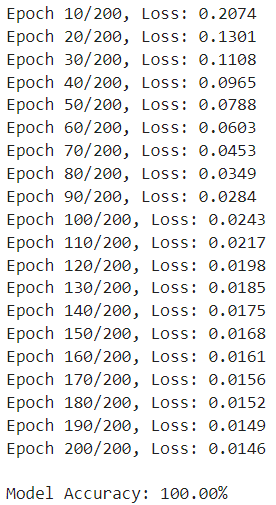
Output

Figure 1

Experiment 5

1. Implementation of Radial Basis Function.

Code Snippet:

import numpy as np

import matplotlib.pyplot as plt

class RBFNN:

def \_\_init\_\_(self, sigma):

self.sigma = sigma

self.centers = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])

self.weights = None

def \_gaussian(self, x, c):

return np.exp(-np.linalg.norm(x - c) \*\* 2 / (2 \* self.sigma \*\* 2))

def \_calculate\_activation(self, X):

activations = np.zeros((X.shape[0], self.centers.shape[0]))

for i, center in enumerate(self.centers):

for j, x in enumerate(X):

activations[j, i] = self.\_gaussian(x, center)

return activations

def fit(self, X, y):

# Calculate activations

activations = self.\_calculate\_activation(X)

# Initialize and solve for weights

self.weights = np.linalg.pinv(activations.T @ activations) @ activations.T @ y

def predict(self, X):

if self.weights is None:

raise ValueError("Model not trained yet. Call fit method first.")

activations = self.\_calculate\_activation(X)

return activations @ self.weights

# Example usage:

if \_\_name\_\_ == "\_\_main\_\_":

# Define XOR dataset

X = np.array([[0.1, 0.1], [0.1, 0.9], [0.9, 0.1], [0.9, 0.9]])

y = np.array([0, 1, 1, 0])

# Initialize and train RBFNN

rbfnn = RBFNN(sigma=0.1)

rbfnn.fit(X, y)

# Predict

predictions = rbfnn.predict(X)

print("Predictions:", predictions)

# Calculate mean squared error

mse = np.mean((predictions - y) \*\* 2)

print("Mean Squared Error:", mse)

# Plot the results

plt.scatter(X[:, 0], X[:, 1], c=predictions, cmap='viridis')

plt.colorbar(label='Predicted Output')

plt.xlabel('X1')

plt.ylabel('X2')

plt.title('RBFN Predictions for XOR ')

plt.show()

Output

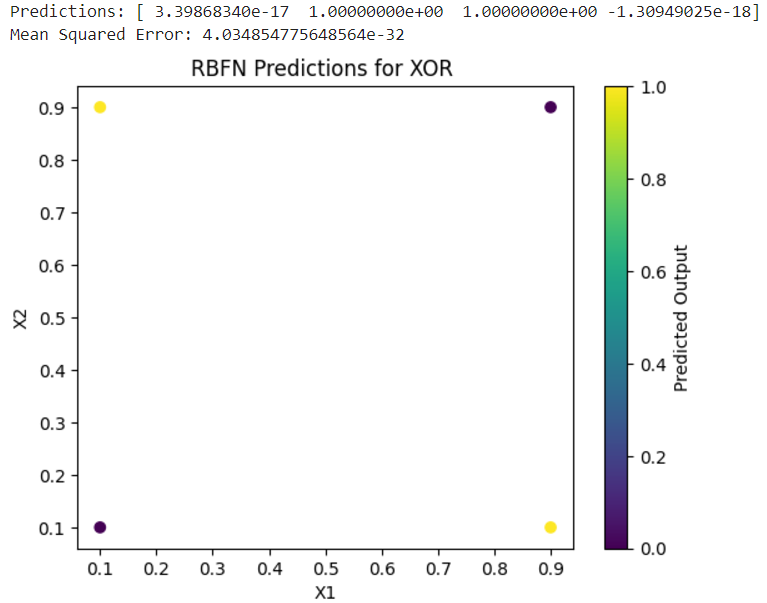


Figure 1

Experiment 6

1. To draw learning curves.

Code Snippet:

import numpy as np

import matplotlib.pyplot as plt

from sklearn.model\_selection import learning\_curve, train\_test\_split

from sklearn.datasets import make\_classification

from sklearn.neural\_network import MLPClassifier

# 1. Generate a synthetic dataset

X, y = make\_classification(

n\_samples=1000, n\_features=20, n\_informative=15, n\_redundant=5,

random\_state=42, n\_classes=2

)

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# 2. Define a simple neural network model

model = MLPClassifier(hidden\_layer\_sizes=(50, 30), max\_iter=500, random\_state=42)

# 3. Compute learning curves

train\_sizes, train\_scores, test\_scores = learning\_curve(

model, X\_train, y\_train, cv=5, scoring='accuracy', train\_sizes=np.linspace(0.1, 1.0, 10)

)

# 4. Calculate mean and standard deviation for train and validation scores

train\_mean = np.mean(train\_scores, axis=1)

train\_std = np.std(train\_scores, axis=1)

test\_mean = np.mean(test\_scores, axis=1)

test\_std = np.std(test\_scores, axis=1)

# 5. Plot learning curves

plt.figure(figsize=(10, 6))

plt.plot(train\_sizes, train\_mean, label="Training Accuracy", color="blue", marker="o")

plt.fill\_between(train\_sizes, train\_mean - train\_std, train\_mean + train\_std, color="blue", alpha=0.2)

plt.plot(train\_sizes, test\_mean, label="Validation Accuracy", color="green", marker="o")

plt.fill\_between(train\_sizes, test\_mean - test\_std, test\_mean + test\_std, color="green", alpha=0.2)

plt.title("Learning Curve")

plt.xlabel("Training Set Size")

plt.ylabel("Accuracy")

plt.legend(loc="best")

plt.grid()

plt.show()

Output

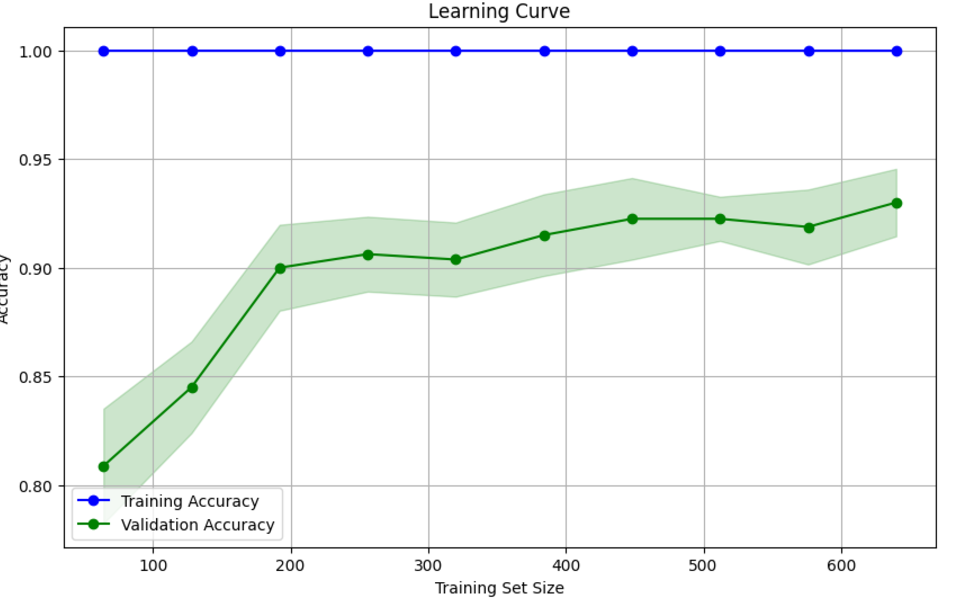


Figure 1

Experiment 7

1. To show the working of Convolutional Neural Network

Code Snippet:

import tensorflow as tf

from tensorflow.keras import layers, models

from tensorflow.keras.datasets import mnist

import matplotlib.pyplot as plt

import numpy as np

# 1. Load and preprocess the MNIST dataset

(X\_train, y\_train), (X\_test, y\_test) = mnist.load\_data()

# Normalize the data to range [0, 1]

X\_train = X\_train.astype('float32') / 255.0

X\_test = X\_test.astype('float32') / 255.0

# Reshape data to include channel dimension (required for CNN)

X\_train = X\_train[..., np.newaxis]

X\_test = X\_test[..., np.newaxis]

# One-hot encode the labels

y\_train = tf.keras.utils.to\_categorical(y\_train, 10)

y\_test = tf.keras.utils.to\_categorical(y\_test, 10)

# 2. Build the CNN model

model = models.Sequential([

layers.Conv2D(32, (3, 3), activation='relu', input\_shape=(28, 28, 1)),

layers.MaxPooling2D((2, 2)),

layers.Conv2D(64, (3, 3), activation='relu'),

layers.MaxPooling2D((2, 2)),

layers.Flatten(),

layers.Dense(128, activation='relu'),

layers.Dense(10, activation='softmax')

])

# 3. Compile the model

model.compile(optimizer='adam',

loss='categorical\_crossentropy',

metrics=['accuracy'])

# 4. Train the model

history = model.fit(X\_train, y\_train, epochs=5, batch\_size=64, validation\_split=0.1)

# 5. Evaluate the model

test\_loss, test\_accuracy = model.evaluate(X\_test, y\_test, verbose=2)

print(f"Test accuracy: {test\_accuracy:.4f}")

# 6. Visualize some predictions

predictions = model.predict(X\_test)

plt.figure(figsize=(10, 10))

for i in range(16):

plt.subplot(4, 4, i + 1)

plt.imshow(X\_test[i].squeeze(), cmap='gray')

plt.title(f"Predicted:{np.argmax(predictions[i])}\nTrue: {np.argmax(y\_test[i])}")

plt.axis('off')

plt.tight\_layout()

plt.show()

# 7. Plot learning curves

plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)

plt.plot(history.history['accuracy'], label='Train Accuracy')

plt.plot(history.history['val\_accuracy'], label='Validation Accuracy')

plt.xlabel('Epochs')

plt.ylabel('Accuracy')

plt.legend()

plt.title('Accuracy Curve')

plt.subplot(1, 2, 2)

plt.plot(history.history['loss'], label='Train Loss')

plt.plot(history.history['val\_loss'], label='Validation Loss')

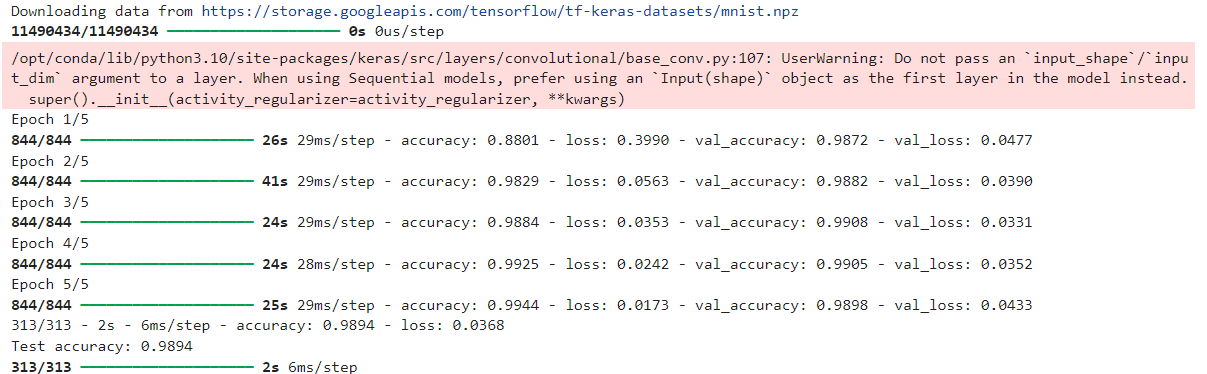
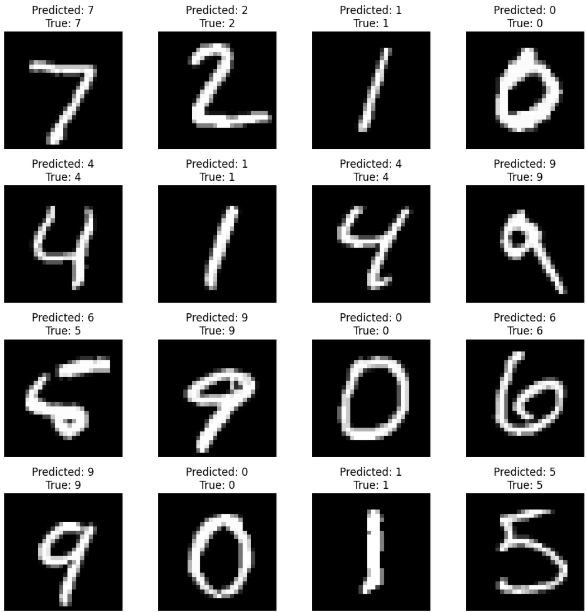
plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.legend()

plt.title('Loss Curve')

plt.show()

**Output**

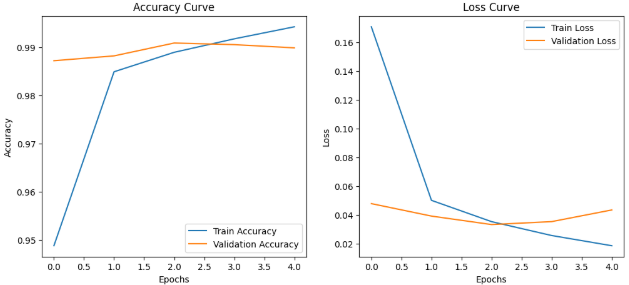


Figure 3

Figure 1

Experiment 8

1. To show the working of Recurrent Neural Network

Code Snippet:

import tensorflow as tf

from tensorflow.keras import layers, models

from tensorflow.keras.datasets import mnist

import matplotlib.pyplot as plt

import numpy as np

# 1. Load and preprocess the MNIST dataset

(X\_train, y\_train), (X\_test, y\_test) = mnist.load\_data()

# Normalize the data to range [0, 1]

X\_train = X\_train.astype('float32') / 255.0

X\_test = X\_test.astype('float32') / 255.0

# Reshape data to include channel dimension (required for CNN)

X\_train = X\_train[..., np.newaxis]

X\_test = X\_test[..., np.newaxis]

# One-hot encode the labels

y\_train = tf.keras.utils.to\_categorical(y\_train, 10)

y\_test = tf.keras.utils.to\_categorical(y\_test, 10)

# 2. Build the CNN model

model = models.Sequential([

layers.Conv2D(32, (3, 3), activation='relu', input\_shape=(28, 28, 1)),

layers.MaxPooling2D((2, 2)),

layers.Conv2D(64, (3, 3), activation='relu'),

layers.MaxPooling2D((2, 2)),

layers.Flatten(),

layers.Dense(128, activation='relu'),

layers.Dense(10, activation='softmax')

])

# 3. Compile the model

model.compile(optimizer='adam',

loss='categorical\_crossentropy',

metrics=['accuracy'])

# 4. Train the model

history = model.fit(X\_train, y\_train, epochs=5, batch\_size=64, validation\_split=0.1)

# 5. Evaluate the model

test\_loss, test\_accuracy = model.evaluate(X\_test, y\_test, verbose=2)

print(f"Test accuracy: {test\_accuracy:.4f}")

# 6. Visualize some predictions

predictions = model.predict(X\_test)

plt.figure(figsize=(10, 10))

for i in range(16):

plt.subplot(4, 4, i + 1)

plt.imshow(X\_test[i].squeeze(), cmap='gray')

plt.title(f"Predicted:{np.argmax(predictions[i])}\nTrue: {np.argmax(y\_test[i])}")

plt.axis('off')

plt.tight\_layout()

plt.show()

# 7. Plot learning curves

plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)

plt.plot(history.history['accuracy'], label='Train Accuracy')

plt.plot(history.history['val\_accuracy'], label='Validation Accuracy')

plt.xlabel('Epochs')

plt.ylabel('Accuracy')

plt.legend()

plt.title('Accuracy Curve')

plt.subplot(1, 2, 2)

plt.plot(history.history['loss'], label='Train Loss')

plt.plot(history.history['val\_loss'], label='Validation Loss')

plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.legend()

plt.title('Loss Curve')

plt.show()

Output

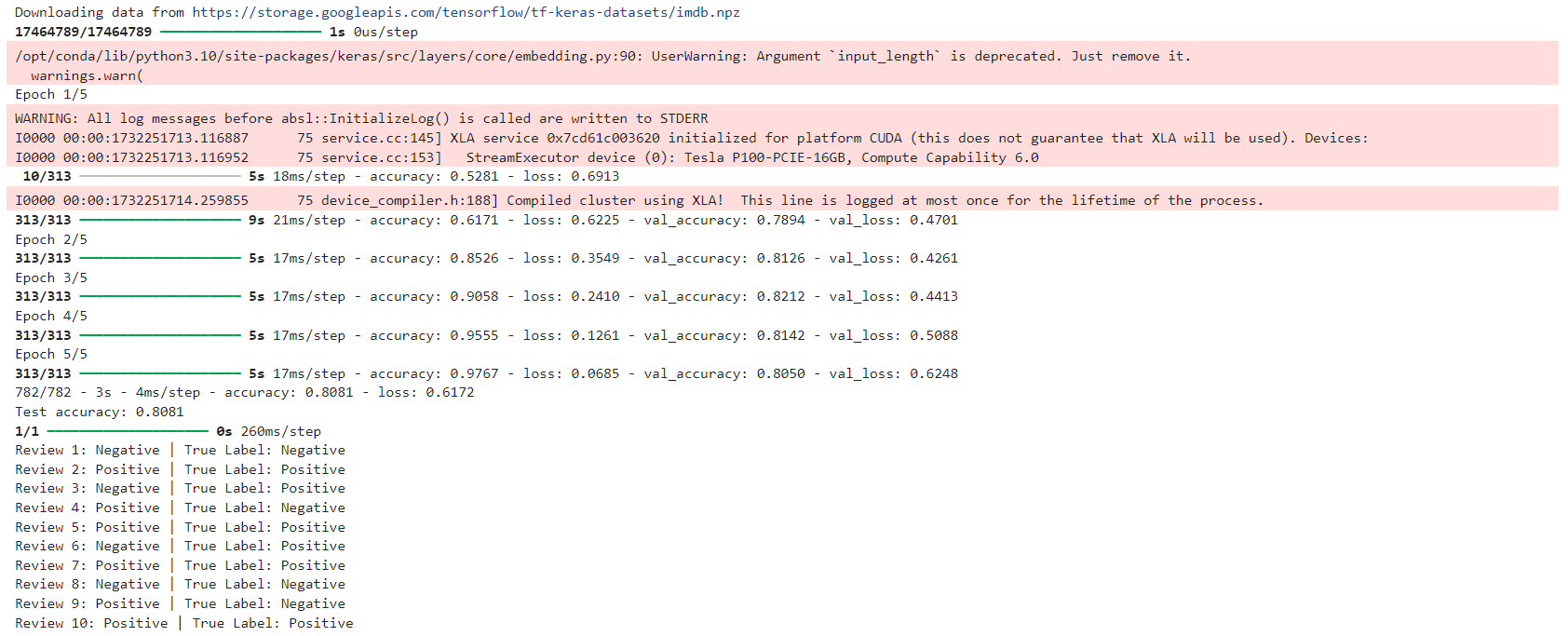


Figure 1

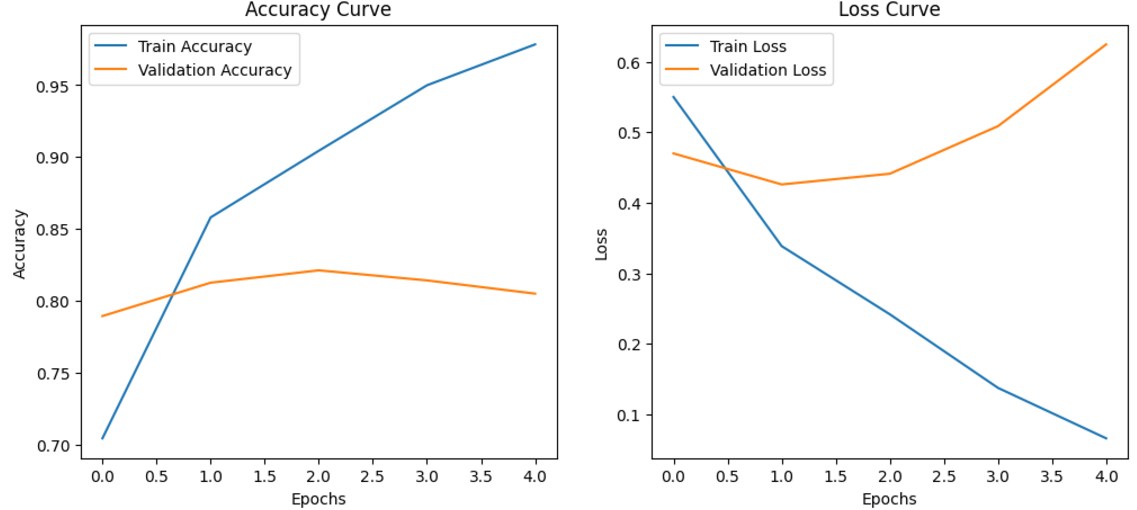


Figure 2

Experiment 9

1. To show the working of Advanced Convolution Neural Network

Code Snippet:

import tensorflow as tf

from tensorflow.keras import layers, models

from tensorflow.keras.datasets import cifar10

from tensorflow.keras.utils import to\_categorical

import matplotlib.pyplot as plt

# 1. Load and preprocess the CIFAR-10 dataset

(X\_train, y\_train), (X\_test, y\_test) = cifar10.load\_data()

# Normalize pixel values to [0, 1] range

X\_train = X\_train.astype('float32') / 255.0

X\_test = X\_test.astype('float32') / 255.0

# One-hot encode labels

y\_train = to\_categorical(y\_train, 10)

y\_test = to\_categorical(y\_test, 10)

# 2. Build the Advanced CNN model

model = models.Sequential([

# First Convolutional Block

layers.Conv2D(32, (3, 3), activation='relu', padding='same', input\_shape=(32, 32, 3)),

layers.BatchNormalization(),

layers.Conv2D(32, (3, 3), activation='relu', padding='same'),

layers.BatchNormalization(),

layers.MaxPooling2D((2, 2)),

layers.Dropout(0.25),

# Second Convolutional Block

layers.Conv2D(64, (3, 3), activation='relu', padding='same'),

layers.BatchNormalization(),

layers.Conv2D(64, (3, 3), activation='relu', padding='same'),

layers.BatchNormalization(),

layers.MaxPooling2D((2, 2)),

layers.Dropout(0.25),

# Third Convolutional Block

layers.Conv2D(128, (3, 3), activation='relu', padding='same'),

layers.BatchNormalization(),

layers.Conv2D(128, (3, 3), activation='relu', padding='same'),

layers.BatchNormalization(),

layers.MaxPooling2D((2, 2)),

layers.Dropout(0.25),

# Fully Connected Layers

layers.GlobalAveragePooling2D(),

layers.Dense(256, activation='relu'),

layers.Dropout(0.5),

layers.Dense(10, activation='softmax') # Output layer for 10 classes

])

# 3. Compile the model

model.compile(optimizer='adam',

loss='categorical\_crossentropy',

metrics=['accuracy'])

# 4. Train the model

history = model.fit(X\_train, y\_train, epochs=15, batch\_size=64, validation\_split=0.2)

# 5. Evaluate the model

test\_loss, test\_accuracy = model.evaluate(X\_test, y\_test, verbose=2)

print(f"Test Accuracy: {test\_accuracy:.4f}")

# 6. Plot learning curves

plt.figure(figsize=(12, 5))

# Accuracy curve

plt.subplot(1, 2, 1)

plt.plot(history.history['accuracy'], label='Train Accuracy')

plt.plot(history.history['val\_accuracy'], label='Validation Accuracy')

plt.xlabel('Epochs')

plt.ylabel('Accuracy')

plt.legend()

plt.title('Accuracy Curve')

# Loss curve

plt.subplot(1, 2, 2)

plt.plot(history.history['loss'], label='Train Loss')

plt.plot(history.history['val\_loss'], label='Validation Loss')

plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.legend()

plt.title('Loss Curve')

plt.show()

# 7. Display predictions

class\_names = ['Airplane', 'Automobile', 'Bird', 'Cat', 'Deer', 'Dog', 'Frog', 'Horse', 'Ship', 'Truck']

predictions = model.predict(X\_test[:10])

plt.figure(figsize=(10, 5))

for i in range(10):

plt.subplot(2, 5, i + 1)

plt.imshow(X\_test[i])

plt.title(f"Pred: {class\_names[predictions[i].argmax()]}\nTrue: {class\_names[y\_test[i].argmax()]}")

plt.axis('off')

plt.tight\_layout()

plt.show()

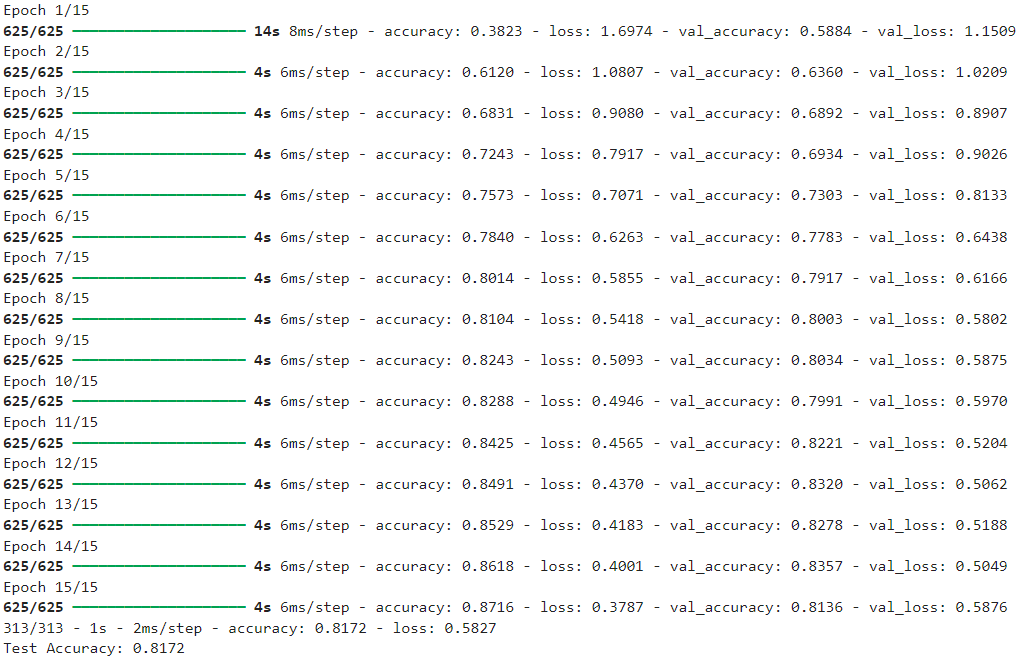
Output

Figure 1

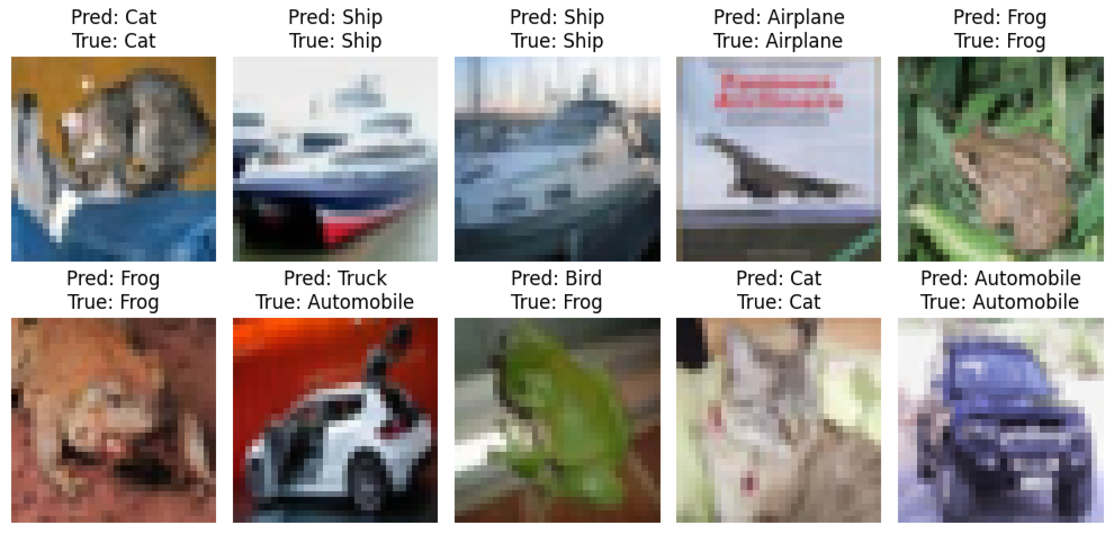
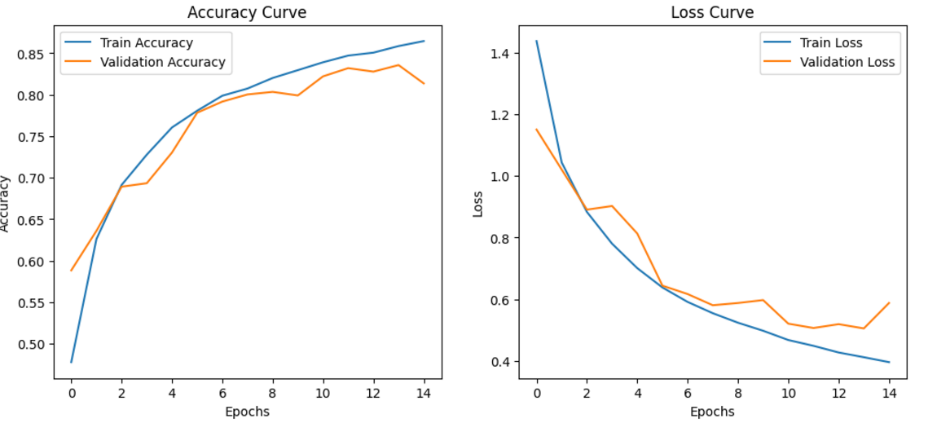


Figure 3

Figure 2

Experiment 10

1. To perform language processing

Code Snippet:

import spacy

import nltk

from nltk.tokenize import word\_tokenize

from nltk.corpus import stopwords

from nltk.stem import PorterStemmer

import matplotlib.pyplot as plt

from wordcloud import WordCloud

from collections import Counter

# Download necessary NLTK resources

nltk.download('punkt')

nltk.download('stopwords')

# Load spaCy model for lemmatization

nlp = spacy.load("en\_core\_web\_sm")

# Sample text

text = """

Natural Language Processing (NLP) is a sub-field of artificial intelligence that focuses on the interaction between

computers and humans through natural language. It involves text analysis, machine translation, sentiment analysis,

and much more. NLP has applications in chatbots, virtual assistants, and automated text processing systems.

"""

# 1. Tokenization using NLTK

tokens = word\_tokenize(text)

print("Tokens:", tokens)

# 2. Stopword Removal using NLTK

stop\_words = set(stopwords.words('english'))

filtered\_tokens = [word for word in tokens if word.lower() not in stop\_words]

print("\nTokens after stopword removal:", filtered\_tokens)

# 3. Stemming using NLTK

stemmer = PorterStemmer()

stemmed\_tokens = [stemmer.stem(word) for word in filtered\_tokens]

print("\nStemmed Tokens:", stemmed\_tokens)

# 4. Lemmatization using spaCy

lemmatized\_tokens = [token.lemma\_ for token in nlp(" ".join(filtered\_tokens)).doc]

print("\nLemmatized Tokens:", lemmatized\_tokens)

# 5. Word Frequency Analysis

word\_counts = Counter(lemmatized\_tokens)

print("\nWord Frequencies:", word\_counts)

# 6. Word Cloud Visualization

wordcloud = WordCloud(width=800, height=400, background\_color='white').generate(" ".join(lemmatized\_tokens))

# Plot Word Cloud

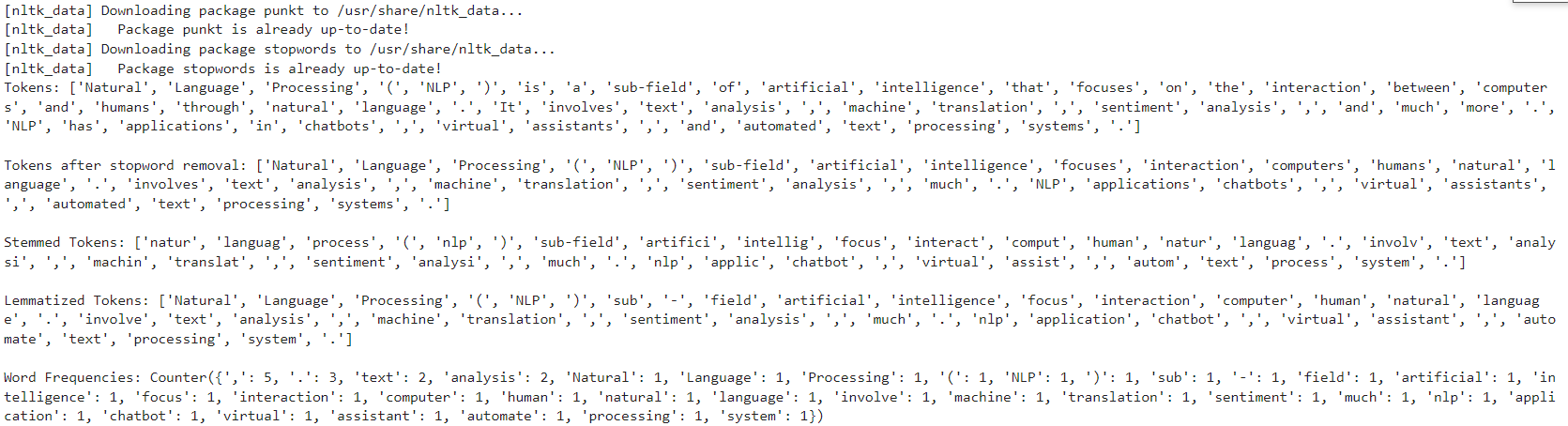
plt.figure(figsize=(10, 5))

plt.imshow(wordcloud, interpolation='bilinear')

plt.axis('off')

plt.title("Word Cloud Visualization", fontsize=16)

plt.show()

Output

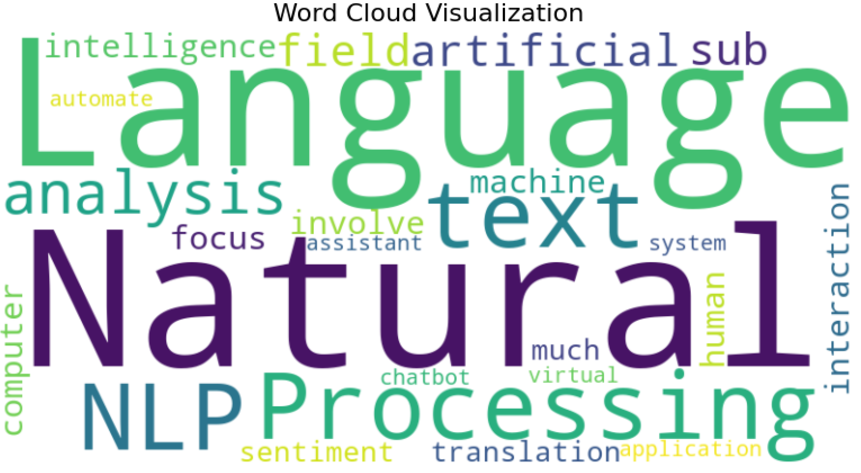


Figure 1

Figure 2