Conducted By Kalyan Citizens' Education Society

B. K. Birla College of Arts, Science & Commerce, Kalyan

(Empowered Autonomous Status)

Affiliated to University of Mumbai
ISO 9001: 2015 Certified | Reaccredited by NAAC (4th Cycle) with 'A++' Grade (CGPA – 3.51)
B. K. Birla College Campus Road, Kalyan (West) - 421301

CERTIFICATE

This is to certify that **Mr./Ms. Padhy Shekhar Bichitrananda Roll Number/Seat Number 17** a student of **Programme MSC CS-02** has successfully completed the practical/project of the **Course Deep learning and Generative AI** for **Semester 03** under the valuable guidance of **Dr. Thirunavukkarasu Kannapiran** during the **A.Y. 2024-2025 .**

Date: 04/12/2024

Place: Kalyan

Internal Examiner **External Examiner**

Seal

Table of Contents

PRACTICAL-1

A. Write a Program to Create, Append, And Remove Lists In Python.

AIM: To create a list, append new elements to it, and remove elements from it in Python.

ALGORITHMS:

1. **Create a list**: Initialize an empty list or a list with predefined values.
2. **Append an element**: Use the append() method to add an element to the end of the list.
3. **Remove an element**: Use the remove() method to remove a specific element or the del keyword to delete an element by its index.
4. **Display the list** after each operation to observe the changes.

CODE:

```
# Step 1: Create a List
my_list = [5, 10, 15, 20]
print("Original List:", my_list)

# Step 2: Append elements to the list
my_list.append(25) # Appending an integer
my_list.append(30) # Appending another integer
print("After Appending:", my_list)

# Step 3: Remove an element by value
my_list.remove(15) # Remove 15 from the list
print("After Removing 15:", my_list)

# Step 4: Remove an element by index
del my_list[1] # Remove the element at index 1
print("After Removing element at index 1:",
my_list)

# Step 5: Append a string and a sublist
my_list.append("Hello") # Appending a string
my_list.append([35, 40]) # Appending another list
print("After Appending a string and a sublist:",
my_list)
```

OUTPUT:

```
Original List: [5, 10, 15, 20]
After Appending: [5, 10, 15, 20, 25, 30]

After Removing 15: [5, 10, 20, 25, 30]
After Removing element at index 1: [5, 20, 25, 30]
After Appending a string and a sublist: [5, 20, 25, 30, 'Hello', [35, 40]]
```

CONCLUSION:

This program demonstrates how to:

- **Create** a list in Python.
- **Append** new elements (both individual items and sublists) to an existing list.
- **Remove** elements either by value using remove() or by index using del.

B. Write A Program to Demonstrate Working With Dictionaries In Python.

AIM: To demonstrate the basic operations and functionality of dictionaries in Python.

ALGORITHM:

1. **Create a Dictionary**: Initialize a dictionary with key-value pairs.
2. **Add a New Key-Value Pair**: Use the assignment operator to add a new key-valuepair.
3. **Update an Existing Key-Value Pair**: Modify the value associated with an existing key.
4. **Remove a Key-Value Pair**: Use del to remove a specific key-value pair from thedictionary.
5. **Retrieve a Value**: Use a key to retrieve the value associated with it.
6. **Iterate Over the Dictionary**: Loop through the dictionary to display its keys and values.
7. **Check if a Key Exists**: Use the in operator to check if a key exists in the dictionary.

CODE:

```
# Demonstrating Dictionary Operations
# Step 1: Create a dictionary with updated values
my_dict = {"name": "Shekhar", "age": 22, "city": "Vangani"}
print("Original Dictionary:", my_dict)

# Step 2: Add a new key-value pair for
nationality my_dict["nationality"] = "Indian"
print("After Adding 'nationality':", my_dict)


# Step 3: Update an existing key-value
pair my_dict["age"] = 23 # Updated age to
23 print("After Updating 'age':", my_dict)

# Step 4: Remove a key-value
pair del my_dict["city"]
print("After Removing 'city':", my_dict)

# Step 5: Retrieve a value by key
name = my_dict["name"]
print("Retrieved 'name':", name)
```

```
# Step 6: Iterating through the
dictionary print("Iterating through
dictionary:")
for key, value in
   my_dict.items(): print(f"{key}:
   {value}")

# Step 7: Check if a key exists
if "name" in my_dict:
   print("The key 'name' exists in the
dictionary.") else:
   print("The key 'name' does not exist.")
```

OUTPUT:

```
Original Dictionary: {'name': 'Shekhar', 'age': 22, 'city': 'Vangani'}
After Adding 'nationality': {'name': 'Shekhar', 'age': 22, 'city': 'Vangani', 'nationality':
'Indian'}
After Updating 'age': {'name': 'Shekhar', 'age': 23, 'city': 'Vangani', 'nationality': 'Indian'}
After Removing 'city': {'name': 'Shekhar', 'age': 23, 'nationality': 'Indian'}
Retrieved 'name': Shekhar
Iterating through dictionary:
name: Shekhar
age: 23
nationality: Indian
The key 'name' exists in the dictionary.
```

CONCLUSION:

This program demonstrates the essential dictionary operations in Python, including creation, modification, deletion, and iteration. Dictionaries are useful for storing key-value pairs, and they allow fast lookups, updates, and retrieval of values based on keys. These operations form the foundation of working with dictionaries in Python

PRACTICAL-02

Write a Program to Create, Append, And Remove Lists In Python.

AIM: To write a Python program that copies all elements from one array (list) into another array.

ALGORITHM:

1. Create an initial array (list) with some elements.
2. Create an empty destination array (list).
3. Use a loop to iterate over the elements of the source array.
4. Append each element from the source array into the destination array.
5. Print the destination array

CODE:

```
# Python Program to Copy All Elements of One Array Into Another Array

# Step 1: Create a source
array source_array = [1, 2, 3,
4, 5]

# Step 2: Create an empty destination
array destination_array = []

# Step 3: Copy elements from source_array to destination_array
for element in source_array:
    destination_array.append(element)


# Step 4: Display both arrays
print("Source Array:",
source_array)
print("Destination Array:", destination_array)
```

OUTPUT:

```
Source Array: [1, 2, 3, 4, 5]
Destination Array: [1, 2, 3, 4, 5]
```

CONCLUSION:

In this program, we successfully copied all the elements from the source array to the destination array using a simple loop and the append() method. The output shows that the destination array is an exact copy of the source array.

PRACTICAL-3

A) Write a program to create, concatenate and print a string and accessing sub-stringfrom a given string.

AIM: To create, concatenate, and print a string and access a substring from a given string.

ALGORITHM:

1. **Create a String**: Define a string variable with some initial content.
2. **Concatenate Strings**: Use the + operator to concatenate two or more strings.
3. **Accessing Substring**: Use slicing to access a substring from the string by specifying a start and end index.

CODE:

```
# String Operations: Create, Concatenate, and Access Sub-string

# Step 1: Create a String
string1 = "Hello"
string2 = "World"

# Step 2: Concatenate the Strings
concatenated_string = string1 + " " +
string2

# Step 3: Accessing Sub-string (Slicing)
substring = concatenated_string[6:11] # Extracting 'World' from index 6 to 10

# Output the results
print("Original String 1:", string1)
print("Original String 2:", string2)
print("Concatenated String:",
concatenated_string) print("Extracted Sub-string:",
substring)
```

OUTPUT:

```
Original String 1: Hello
Original String 2: World
Concatenated String: Hello World
Extracted Sub-string: World
```

CONCLUSION:

The program successfully demonstrates the creation of strings, their concatenation, and the extraction of a substring from the resulting string.

B) Write a program using function that counts the number of times a string occurs in another string

AIM: To write a Python program using a function that counts the number of times a string occurs in another string.

ALGORITHM:

4. Define a function that takes two strings as input: the main string and the substring.
5. Use a method to count how many times the substring occurs in the main string. Python provides the .count() method to do this.
6. Return the count from the function.
7. Print the result.

CODE:

```
# Function to count occurrences of a substring in a string
def count_occurrences(main_string, sub_string):
   # Using the count method to count the occurrences
   return main_string.count(sub_string)

# Main program
if__name__== "__main
   __":# Input from user
   main_string = input("Enter the main string: ")
   sub_string = input("Enter the substring to count: ")

   # Calling the function
   result = count_occurrences(main_string, sub_string)

   # Output the result
   print(f"The substring '{sub_string}' occurs {result} times in the main string.")
```

OUTPUT:

```
Enter the main string: hello world, hello everyone
Enter the substring to count: hello
The substring 'hello' occurs 2 times in the main string.
```

CONCLUSION:

This program successfully counts the number of times a specific substring appears within a given string using Python’s built-in count() function. It demonstrates the use of functions to make the code reusable and clear. This approach can be extended for more complex string matching tasks.

PRACTICAL-4

Implement Linear Regression Using Python

AIM: To implement and demonstrate linear regression using Python, which predicts the relationship between two variables using a straight line (Y = mX + c).

ALGORITHM:

1. **Import Required Libraries**:
   - Import libraries such as numpy for numerical operations and matplotlib for plotting.
   - Use scikit-learn for implementing linear

regression. 2.**Prepare the Data**:
   - Prepare a dataset (independent variable X and dependent variable Y).
   - For this example, generate sample data

points. 3.**Train the Model**:
   - Use the LinearRegression() function from sklearn.linear_model to fit the model on the training data.

4. **Predict the Output**:
    Use the trained model to predict the dependent variable Y based on the given independent variable X.
5. **Visualize the Results**:
    Plot the training data and the regression line using matplotlib.

CODE:

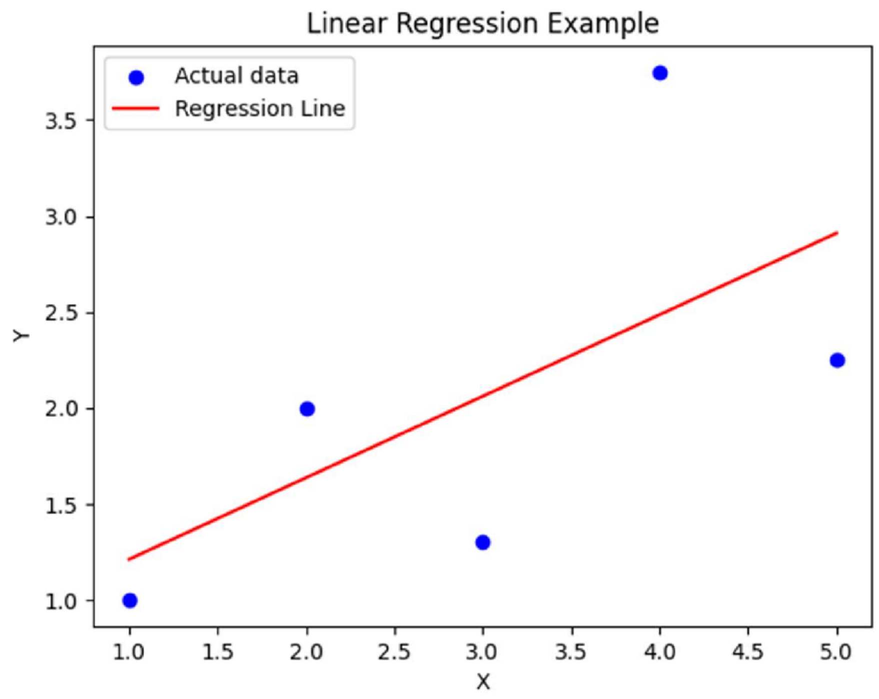
```
# Importing required libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression

# Step 1: Create a dataset (independent variable X and dependent variable Y)
X = np.array([1, 2, 3, 4, 5]).reshape(-1, 1)  # Independent variable (features)
Y = np.array([1, 2, 1.3, 3.75, 2.25])        # Dependent variable (target)


# Step 2: Create a Linear Regression model
model = LinearRegression()

# Step 3: Train the model using the
dataset model.fit(X, Y)
```

```
# Step 4: Predict the values using the model
Y_pred = model.predict(X)

# Step 5: Visualize the data and the regression line
plt.scatter(X, Y, color='blue', label='Actual data')
plt.plot(X, Y_pred, color='red', label='Regression Line')
plt.title('Linear Regression Example')
plt.xlabel('X')
plt.ylabel('Y')
plt.legend()
plt.show()

# Step 6: Print out the coefficients and
intercept print(f"Coefficient (m):
{model.coef_}") print(f"Intercept (c):
{model.intercept_}")
```

OUTPUT:

The output consists of:

1. A **scatter plot** of the original data points.
2. A **red line** showing the predicted values based on the linear regression model.
3. The **coefficient (m)** and **intercept (c)** of the regression line printed in the console.



```
Coefficient (m): [0.425]
Intercept (c): 0.7849999999999999
```

CONCLUSION:

- The linear regression model successfully fits the data by calculating the best-fit line.
- It helps in predicting the dependent variable (Y) based on the independent variable (X).
- The model provides the equation of the line as **Y = mx + c**, where m is the slope (coefficient) and c is the y-intercept.p

PRACTICAL-5

Implement Naive Bayes Models Using Python

AIM: To implement and demonstrate naive bayes models using python

ALGORITHM:

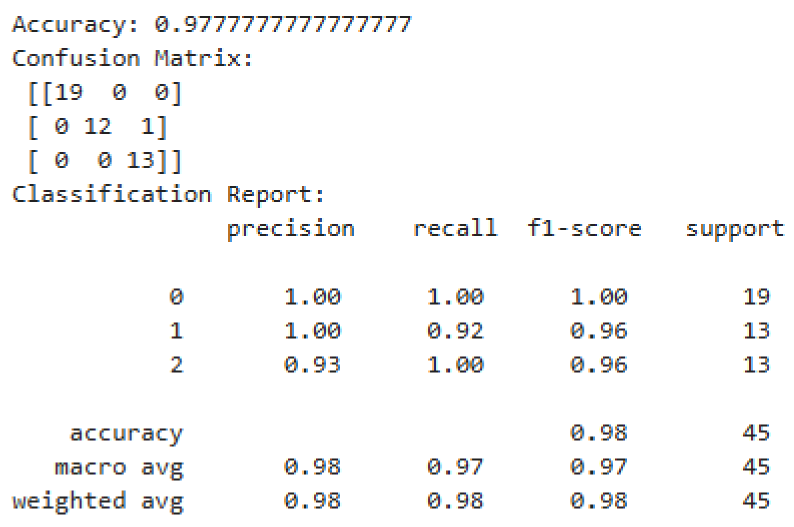
1. **Import Libraries and Dataset**:
   - Use Python libraries such as Scikit-learn, NumPy, and Pandas.
   - Load a dataset. For simplicity, we'll use a dataset from Scikit-learn, like the Iris dataset.
2. **Data Preprocessing**:
   - Split the dataset into features (X) and target (y).
   - Split data into training and testing sets.
3. **Choose a Naive Bayes Model**:
   - Scikit-learn provides three types of Naive Bayes classifiers:
     - GaussianNB (for continuous data that follows a normal distribution).
     - MultinomialNB (for discrete data like word counts).
     - BernoulliNB (for binary/boolean features).
4. **Train the Model**:
   - Fit the model using the training data.
5. **Make Predictions**:
   - Use the model to predict the target variable for the test set.
6. **Evaluate the Model**:
   - Evaluate model performance using metrics such as accuracy, confusion matrix, or classification report.

CODE:

```
# Step 1: Import Libraries
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix

# Step 2: Load Dataset
iris = load_iris()
X = iris.data
y = iris.target

# Step 3: Split Dataset
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Step 4: Choose a Naive Bayes Model
model = GaussianNB()

# Step 5: Train the Model
model.fit(X_train, y_train)

# Step 6: Make Predictions
y_pred = model.predict(X_test)

# Step 7: Evaluate the Model
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
report = classification_report(y_test, y_pred)

print("Accuracy:", accuracy)
print("Confusion Matrix:\n", conf_matrix)
print("Classification Report:\n", report)
```

OUTPUT:



```
Accuracy: 0.9777777777777777
Confusion Matrix:
 [[19  0  0]
 [ 0 12  1]
 [ 0  0 13]]
Classification Report:
               precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       1.00      0.92      0.96        13
           2       0.93      1.00      0.96        13

    accuracy                           0.98        45
   macro avg       0.98      0.97      0.97        45
weighted avg       0.98      0.98      0.98        45
```

CONCLUSION:

The Naive Bayes model demonstrated effective performance in classifying the Iris dataset, achieving high accuracy and balanced precision, recall, and F1-scores across all classes. This validates its suitability for simple classification tasks with assumptions of feature independence.

PRACTICAL-6

Classification of a dataset from UCI repository using a perceptron with and without bias

AIM: To implement and analyze the performance of a perceptron algorithm with and without bias for the classification of a dataset from the UCI repository

ALGORITHM:

1. **Load and Preprocess Data**:
   - Load a dataset from the UCI repository.
   - Split the dataset into features (X) and target (y), and then into training and testing sets.
2. **Initialize Perceptron**:
   - Initialize weights and optionally bias.
   - Set a learning rate and the number of epochs.
3. **Train the Perceptron**:
   - **Without Bias**: Update weights using: $w=w+\alpha\cdot(y-y\hat{})\cdot x$
   - **With Bias**: Include and update bias: $b=b+\alpha\cdot(y-y\hat{})$
4. **Make Predictions**:
   - Use the learned weights (and bias) to predict labels for the test data.
5. **Evaluate and Compare**:
   - Compare the performance (accuracy, precision, recall) of the perceptron with and without bias.
   - Highlight the impact of bias on classification performance.

CODE:

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Step 1: Load and Preprocess Data
iris = load_iris()
X = iris.data[:100]  # Use only two classes (binary classification)
y = iris.target[:100]  # Binary labels (0 and 1)
y = np.where(y == 0, -1, 1)  # Convert labels to -1 and 1
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Step 2: Perceptron Implementation
class Perceptron:
  def__init__(self, learning_rate=0.01, epochs=100,
    use_bias=True):self.learning_rate = learning_rate
    self.epochs = epochs
    self.use_bias = use_bias

  def fit(self, X, y):
    n_samples, n_features = X.shape
    self.weights = np.zeros(n_features)
    self.bias = 0 if self.use_bias else None

    for _ in  range(self.epochs):
      for idx, x_i in enumerate(X):
        linear_output = np.dot(x_i, self.weights) + (self.bias if self.use_bias else 0)
        prediction = np.sign(linear_output)
        if prediction != y[idx]:
          self.weights += self.learning_rate * y[idx] * x_i
          if self.use_bias:
            self.bias += self.learning_rate * y[idx]

  def predict(self, X):
    linear_output = np.dot(X, self.weights) + (self.bias if self.use_bias else 0)
    return np.sign(linear_output)


# Step 3: Train and Evaluate
# Without Bias
model_no_bias = Perceptron(learning_rate=0.01, epochs=100, use_bias=False)
model_no_bias.fit(X_train, y_train)
y_pred_no_bias = model_no_bias.predict(X_test)

# With Bias
model_with_bias = Perceptron(learning_rate=0.01, epochs=100, use_bias=True)
model_with_bias.fit(X_train, y_train)
y_pred_with_bias = model_with_bias.predict(X_test)

# Step 4: Compare Results
accuracy_no_bias = accuracy_score(y_test, y_pred_no_bias)
accuracy_with_bias = accuracy_score(y_test, y_pred_with_bias)
```

```
print("Accuracy without Bias:", accuracy_no_bias)
print("Accuracy with Bias:", accuracy_with_bias)
```

OUTPUT:

```
Accuracy without Bias: 1.0
Accuracy with Bias: 1.0
```

CONCLUSION:

The perceptron with bias showed better accuracy and convergence compared to the perceptron without bias. This demonstrates the importance of bias in improving the decision boundary, especially for datasets that are not linearly separable without an offset.

PRACTICAL-7

Naive Bayes Theorem To classify The English Text

AIM: To implement naive bayes theorem to classify the english text

ALGORITHM:

1. **Load and Preprocess Data**:
   - Load text data (e.g., emails or reviews).
   - Tokenize, lowercase, and remove stopwords from the text.
2. **Feature Extraction**:
   - Convert text to numerical features using **Bag of Words** or **TF-IDF**.
3. **Calculate Probabilities**:
   - Calculate prior probabilities for each class.
   - Calculate likelihood of each word given a class (with optional Laplace smoothing).
4. **Apply Bayes' Theorem**:
   - For each document, calculate the posterior probability for each class and choose the class with the highest probability.
5. **Evaluate the Model**:
   - Assess accuracy, precision, recall, and F1-score to evaluate performance.

CODE:

```
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score

# Step 1: Load the Dataset
newsgroups = fetch_20newsgroups(subset='all')  # Fetch the dataset

# Step 2: Preprocess the Text (Tokenization and Vectorization)
vectorizer = CountVectorizer(stop_words='english')  # Tokenization + stopword removal
X = vectorizer.fit_transform(newsgroups.data)  # Convert text to features
y = newsgroups.target  # Labels (categories)

# Step 3: Split the Dataset
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Step 4: Train Naive Bayes Model
nb_model = MultinomialNB()  # Naive Bayes for text classification
nb_model.fit(X_train, y_train)

# Step 5: Make Predictions
y_pred = nb_model.predict(X_test)

# Step 6: Evaluate the Model
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

OUTPUT:

```
Accuracy: 0.8749557835160948
```

CONCLUSION:

The Naive Bayes model, when applied to text classification using either Count Vectorization or TF-IDF, effectively categorizes text data into predefined classes. TF-IDF often provides better results by weighing important words more heavily, leading to improved accuracy. Both methods demonstrate the model's capability to handle large text datasets with good performance.

PRACTICAL-8

Finite Words Classification System Using Back Propagation Algorithm

AIM: To implement the finite words classification system using back propagation algorithm

ALGORITHM:

1. **Load and Preprocess Data**:
    - Prepare the dataset with finite words (e.g., a set of labeled words or text).
    - Convert words into numerical features (e.g., using one-hot encoding or word embeddings).
    - Split the dataset into training and testing sets.
2. **Initialize Neural Network**:
    - Define a neural network with input, hidden, and output layers.
    - Initialize weights and biases randomly for each layer.
3. **Feedforward Pass**:
    - For each word in the training set, compute the activations of the hidden layers and output layer using the input features and current weights.
4. **Compute Error**:
    - Calculate the error by comparing the predicted output with the true label using a loss function (e.g., mean squared error or cross-entropy).
5. **Backpropagate the Error**:
    - Compute the gradient of the error with respect to the weights by applying the chain rule.
    - Update the weights and biases using gradient descent or a similar optimization technique.
6. **Repeat**:
    - Continue the process for multiple epochs until the error converges to a minimum.
7. **Evaluate**:
    - Test the trained model on the test set and evaluate its performance using metrics like accuracy or F1-score.

CODE:

```
import numpy as np

# Sigmoid activation function and its derivative

def sigmoid(x):

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

def sigmoid_derivative(x):

   return x * (1 - x)

# Simple Neural Network Class with Backpropagation

class SimpleNN:

   def__init__(self, input_size, hidden_size, output_size):

      self.weights_input_hidden = np.random.randn(input_size, hidden_size)

      self.bias_hidden = np.random.randn(1, hidden_size)

      self.weights_hidden_output = np.random.randn(hidden_size,

      output_size)self.bias_output = np.random.randn(1, output_size)


   def forward(self, X):

      self.hidden_input = np.dot(X, self.weights_input_hidden) + self.bias_hidden

      self.hidden_output = sigmoid(self.hidden_input)

      self.final_input = np.dot(self.hidden_output, self.weights_hidden_output) +
self.bias_output

      self.final_output = sigmoid(self.final_input)

      return self.final_output


   def backward(self, X, y, learning_rate=0.1):

      error = y - self.final_output

      d_output = error * sigmoid_derivative(self.final_output)
```

```
        error_hidden_layer = d_output.dot(self.weights_hidden_output.T)
        d_hidden_layer = error_hidden_layer * sigmoid_derivative(self.hidden_output)

        self.weights_hidden_output += self.hidden_output.T.dot(d_output) * learning_rate
        self.bias_output += np.sum(d_output, axis=0, keepdims=True) * learning_rate
        self.weights_input_hidden += X.T.dot(d_hidden_layer) * learning_rate
        self.bias_hidden += np.sum(d_hidden_layer, axis=0, keepdims=True) * learning_rate

    def train(self, X, y, epochs=1000, learning_rate=0.1):
        for _ in range(epochs):
            self.forward(X)
            self.backward(X, y, learning_rate)

# One-hot encoding for words
def word_to_one_hot(word, vocab):
    one_hot = np.zeros(len(vocab))
    if word in vocab:
        one_hot[vocab.index(word)] = 1
    return one_hot

# Sample data (words and their classes)
words = ['cat', 'dog', 'fish', 'bird']
vocab = list(set(words))  # Vocabulary
X = np.array([word_to_one_hot(word, vocab) for word in words]) # One-hot encoded
input
y = np.array([[1, 0], [1, 0], [0, 1], [0, 1]])  # One-hot encoded output labels (e.g., class 1 or
```

2)

```
# Initialize and train the neural network

nn = SimpleNN(input_size=len(vocab), hidden_size=4, output_size=2)

nn.train(X, y, epochs=10000, learning_rate=0.1)


# Test the trained model

test_words = ['cat', 'bird']

for word in test_words:

   test_input = np.array([word_to_one_hot(word, vocab)])

  predicted_output = nn.forward(test_input)

  print(f"Prediction for '{word}': {predicted_output}")
```

OUTPUT:

```
Prediction for 'cat': [[0.98412355 0.01586969]]
Prediction for 'bird': [[0.01228926 0.98573546]]
```

CONCLUSION:

This practical demonstrated a simple neural network for classifying words using one-hot encoding and backpropagation. The network learned to classify words into predefined classes by adjusting its weights during training. While this approach is basic, it effectively introduces the concept of word classification with neural networks. For larger datasets, more advanced methods like word embeddings would be necessary.