NLP LAB MANUAL

1.Write a Python program for the following preprocessing of text in NLP:
● Tokenization
● Filtration
● Script Validation
● Stop Word Removal
● Stemming

```
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import PorterStemmer
import re

# Download required NLTK resources (only run once)
nltk.download('punkt')
nltk.download('stopwords')

# Sample text
text = "Hello there! This is a simple example to demonstrate text preprocessing in NLP. It will
include tokenization, stop word removal, and stemming."

# 1. Tokenization
def tokenize(text):
    return word_tokenize(text)

# 2. Filtration: Remove non-alphabetic tokens
def filter_non_alpha(tokens):
    return [word for word in tokens if word.isalpha()]

# 3. Script Validation: Remove non-ASCII characters
def validate_script(tokens):
    return [word for word in tokens if all(ord(c) < 128 for c in word)]

# 4. Stop Word Removal
def remove_stopwords(tokens):
    stop_words = set(stopwords.words('english'))
    return [word for word in tokens if word.lower() not in stop_words]

# 5. Stemming (using Porter Stemmer)
def stemming(tokens):
    stemmer = PorterStemmer()
    return [stemmer.stem(word) for word in tokens]

# Full Preprocessing Function
def preprocess(text):
    tokens = tokenize(text)
    tokens = filter_non_alpha(tokens)
    tokens = validate_script(tokens)
    tokens = remove_stopwords(tokens)
    tokens = stemming(tokens)
    return tokens
```

```
# Preprocess the sample text
preprocessed_text = preprocess(text)

# Print the results
print("Original Text: ", text)
print("Preprocessed Text: ", preprocessed_text)
```

OUTPUT:-

```
Original Text:  Hello there! This is a simple example to demonstrate text
preprocessing in NLP. It will include tokenization, stop word removal, and
stemming.
Preprocessed Text:  ['hello', 'simpl', 'exampl', 'demonstr', 'text',
'preprocess', 'nlp', 'includ', 'token', 'stop', 'word', 'remov', 'stem']
```

2.Demonstrate the N-gram modeling to analyze and establish the probability distribution across
sentences and explore the utilization of unigrams, bigrams, and trigrams in diverse English sentences to illustrate the impact of varying n-gram orders on the calculated probabilities.

```
pip install nltk
import nltk
from nltk.util import ngrams
from collections import Counter

# Download NLTK resources
nltk.download('punkt')

# Sample sentences for analysis
sentences = [
    "I am learning Python programming.",
    "Python is a powerful programming language.",
    "I enjoy solving problems using Python."
]

# Function to calculate n-grams and their probabilities
def calculate_ngram_probabilities(sentences, n):
    # Tokenize sentences into words
    words = [nltk.word_tokenize(sentence.lower()) for sentence in sentences]

    # Generate n-grams
    ngram_list = []
    for sentence in words:
        ngram_list.extend(list(ngrams(sentence, n)))

    # Count n-grams frequencies
    ngram_freq = Counter(ngram_list)

    # Total number of n-grams
    total_ngrams = sum(ngram_freq.values())

    # Calculate probabilities for each n-gram
```

```
    ngram_probabilities = {ngram: freq / total_ngrams for ngram, freq in ngram_freq.items()}

    return ngram_probabilities

# Function to display the results
def display_ngram_results(sentences):
    for n in range(1, 4):  # Unigrams (1), Bigrams (2), Trigrams (3)
        print(f"\n{n}-gram Model Probabilities:")
        probabilities = calculate_ngram_probabilities(sentences, n)

        for ngram, prob in probabilities.items():
            print(f"{' '.join(ngram)} : {prob:.4f}")

# Display results for unigrams, bigrams, and trigrams
display_ngram_results(sentences)
```

OUTPUT:-

```
1-gram Model Probabilities:
i : 0.2500
am : 0.0833
learning : 0.0833
python : 0.2500
programming : 0.0833
is : 0.0833
a : 0.0833
powerful : 0.0833
language : 0.0833
enjoy : 0.0833
solving : 0.0833
problems : 0.0833
using : 0.0833

2-gram Model Probabilities:
i am : 0.0833
am learning : 0.0833
learning python : 0.0833
python programming : 0.0833
is a : 0.0833
a powerful : 0.0833
powerful programming : 0.0833
programming language : 0.0833
i enjoy : 0.0833
enjoy solving : 0.0833
solving problems : 0.0833
problems using : 0.0833
using python : 0.0833

3-gram Model Probabilities:
i am learning : 0.0833
am learning python : 0.0833
learning python programming : 0.0833
python programming is : 0.0833
programming is a : 0.0833
```

is a powerful : 0.0833
a powerful programming : 0.0833
powerful programming language : 0.0833
i enjoy solving : 0.0833
enjoy solving problems : 0.0833
solving problems using : 0.0833
problems using python : 0.0833

3.Investigate the Minimum Edit Distance (MED) algorithm and its application in string comparison and the goal is to understand how the algorithm efficiently computes the minimum
number of edit operations required to transform one string into another.

- **Test the algorithm on strings with different type of variations (e.g., typos, substitutions, insertions, deletions)**
- **Evaluate its adaptability to different types of input variations**

```
# Function to calculate the minimum edit distance (Levenshtein Distance)
def levenshtein_distance(str1, str2):
    # Create a matrix to store the distance between substrings
    len_str1 = len(str1) + 1
    len_str2 = len(str2) + 1
    matrix = [[0] * len_str2 for _ in range(len_str1)]

    # Initialize the matrix
    for i in range(len_str1):
        matrix[i][0] = i
    for j in range(len_str2):
        matrix[0][j] = j

    # Fill the matrix
    for i in range(1, len_str1):
        for j in range(1, len_str2):
            if str1[i-1] == str2[j-1]:
                cost = 0
            else:
                cost = 1

            matrix[i][j] = min(matrix[i-1][j] + 1,       # Deletion
                               matrix[i][j-1] + 1,       # Insertion
                               matrix[i-1][j-1] + cost)  # Substitution

    return matrix[len_str1 - 1][len_str2 - 1]


# Function to test the algorithm with different string variations
def test_levenshtein():
    test_cases = [
        ("kitten", "sitting"),       # Substitution, insertion
        ("flaw", "lawn"),            # Substitution
        ("sunday", "saturday"),      # Insertion, substitution
        ("intention", "execution"),  # Substitution, insertion, deletion
```

```
        ("hello", "hallo"),           # Substitution (typo)
        ("apple", "appl"),            # Deletion
        ("abcde", "fghij")            # Substitution
    ]

    for str1, str2 in test_cases:
        print(f"Comparing: {str1} and {str2}")
        distance = levenshtein_distance(str1, str2)
        print(f"Minimum Edit Distance: {distance}\n")


# Run tests on various string variations
test_levenshtein()
```

OUTPUT:-

```
Comparing: kitten and sitting
Minimum Edit Distance: 3

Comparing: flaw and lawn
Minimum Edit Distance: 2

Comparing: sunday and saturday
Minimum Edit Distance: 3

Comparing: intention and execution
Minimum Edit Distance: 5

Comparing: hello and hallo
Minimum Edit Distance: 1

Comparing: apple and appl
Minimum Edit Distance: 1

Comparing: abcde and fghij
Minimum Edit Distance: 5
```

4.Write a program to implement top-down and bottom-up parser using appropriate context free
grammar.

For understanding S -> E
E -> E + T | E - T | T
T -> T * F | T / F | F
F -> (E) | id

```
# Sample tokens: (identifier: "id", operators: +, -, *, /, parentheses: ( and ))

# Top-Down Parser (Recursive Descent Parser)
```

```
class TopDownParser:
    def __init__(self, tokens):
        self.tokens = tokens
        self.position = 0

    def parse(self):
        return self.S()

    def match(self, expected_token):
        if self.position < len(self.tokens) and self.tokens[self.position] == expected_token:
            self.position += 1
        else:
            raise SyntaxError(f"Expected {expected_token}, found {self.tokens[self.position]}")

    def S(self):
        # S -> E
        return self.E()

    def E(self):
        # E -> E + T | E - T | T
        result = self.T()
        while self.position < len(self.tokens) and self.tokens[self.position] in ('+', '-'):
            self.match(self.tokens[self.position])  # Match '+' or '-'
            result = self.T()  # Parse the next term after + or -
        return result

    def T(self):
        # T -> T * F | T / F | F
        result = self.F()
        while self.position < len(self.tokens) and self.tokens[self.position] in ('*', '/'):
            self.match(self.tokens[self.position])  # Match '*' or '/'
            result = self.F()  # Parse the next factor after * or /
        return result

    def F(self):
        # F -> ( E ) | id
        if self.tokens[self.position] == '(':
            self.match('(')  # Match '('
            result = self.E()  # Parse expression inside parentheses
            self.match(')')  # Match ')'
        else:
            result = self.match('id')  # Match identifier
        return result


# Bottom-Up Parser (Shift-Reduce Parser)
class BottomUpParser:
    def __init__(self, tokens):
        self.tokens = tokens
        self.stack = []

    def parse(self):
```

```
        while self.tokens:
            self.shift()  # Move token to the stack
            self.reduce()  # Apply reduction rules to the stack
        return self.stack

    def shift(self):
        if self.tokens:
            self.stack.append(self.tokens.pop(0))

    def reduce(self):
        # Reduce based on the rules: E -> E + T, T -> T * F, etc.
        # Try to apply the following reductions:
        # E -> E + T
        if len(self.stack) >= 3 and self.stack[-2] == '+' and self.stack[-3] == 'E' and self.stack[-1] ==
'T':
            self.stack = self.stack[:-3] + ['E']
        # E -> E - T
        elif len(self.stack) >= 3 and self.stack[-2] == '-' and self.stack[-3] == 'E' and self.stack[-1] ==
'T':
            self.stack = self.stack[:-3] + ['E']
        # T -> T * F
        elif len(self.stack) >= 3 and self.stack[-2] == '*' and self.stack[-3] == 'T' and self.stack[-1] ==
'F':
            self.stack = self.stack[:-3] + ['T']
        # T -> T / F
        elif len(self.stack) >= 3 and self.stack[-2] == '/' and self.stack[-3] == 'T' and self.stack[-1] ==
'F':
            self.stack = self.stack[:-3] + ['T']
        # F -> id
        elif len(self.stack) >= 1 and self.stack[-1] == 'id':
            self.stack = self.stack[:-1] + ['F']
        # F -> ( E )
        elif len(self.stack) >= 3 and self.stack[-3] == '(' and self.stack[-2] == 'E' and self.stack[-1] ==
')':
            self.stack = self.stack[:-3] + ['F']

# Example Input
tokens = ['id', '+', 'id', '*', 'id']

# Test the Top-Down Parser
print("Top-Down Parsing:")
top_down_parser = TopDownParser(tokens.copy())
try:
    top_down_parser.parse()
    print("Parsing successful!")
except SyntaxError as e:
    print("Parsing failed:", e)

# Test the Bottom-Up Parser
print("\nBottom-Up Parsing:")
bottom_up_parser = BottomUpParser(tokens.copy())
result = bottom_up_parser.parse()
```

```
print("Stack after parsing:", result)
```

OUTPUT:-

```
Top-Down Parsing:
Parsing successful!

Bottom-Up Parsing:
Stack after parsing: ['E']
```

5.Given the following short movie reviews, each labeled with a genre, either comedy or action:

- **fun, couple, love, love comedy**
- **fast, furious, shoot action**
- **couple, fly, fast, fun, fun comedy**
- **furious, shoot, shoot, fun action**
- **fly, fast, shoot, love action and**

A new document D: fast, couple, shoot, fly
Compute the most likely class for D. Assume a Naive Bayes classifier and use add-1 smoothing for the likelihoods.

```
from collections import Counter

# Training data: each tuple contains words and the associated genre (comedy or action)
train_data = [
    (["fun", "couple", "love", "love"], "comedy"),
    (["fast", "furious", "shoot"], "action"),
    (["couple", "fly", "fast", "fun", "fun"], "comedy"),
    (["furious", "shoot", "shoot", "fun"], "action"),
    (["fly", "fast", "shoot", "love"], "action")
]

# New document D (unlabeled, we want to predict the genre)
new_document = ["fast", "couple", "shoot", "fly"]

# Step 1: Create vocabulary and word counts for each class
class_word_counts = {
    "comedy": Counter(),
    "action": Counter()
}
class_counts = {"comedy": 0, "action": 0}

# Fill the counts
for words, genre in train_data:
    class_word_counts[genre].update(words)
    class_counts[genre] += len(words)

# Vocabulary (unique words in the training data)
vocabulary = set(word for words, _ in train_data for word in words)
V = len(vocabulary)  # Vocabulary size

# Step 2: Compute prior probabilities P(class)
total_docs = len(train_data)
```

```
P_comedy = class_counts["comedy"] / total_docs
P_action = class_counts["action"] / total_docs

# Step 3: Add-1 smoothing for word likelihoods P(word | class)
def word_likelihood(word, genre):
    word_count_in_class = class_word_counts[genre][word]
    total_word_count_in_class = class_counts[genre]
    return (word_count_in_class + 1) / (total_word_count_in_class + V)

# Step 4: Compute posterior for each class
def compute_posterior(new_document, genre, prior):
    posterior = prior
    for word in new_document:
        likelihood = word_likelihood(word, genre)
        posterior *= likelihood
    return posterior

# Compute posteriors for both classes
posterior_comedy = compute_posterior(new_document, "comedy", P_comedy)
posterior_action = compute_posterior(new_document, "action", P_action)

# Step 5: Choose the class with the highest posterior
if posterior_comedy > posterior_action:
    print("The most likely genre for the document is Comedy.")
else:
    print("The most likely genre for the document is Action.")
```

OUTPUT:-

The most likely genre for the document is Action.

6.Demonstrate the following using appropriate programming tool which illustrates the use of information retrieval in NLP:

● Study the various Corpus – Brown, Inaugural, Reuters, udhr with various methods like filelds, raw, words, sents, categories
● Create and use your own corpora (plaintext, categorical)
● Study Conditional frequency distributions
● Study of tagged corpora with methods like tagged_sents, tagged_words
● Write a program to find the most frequent noun tags
● Map Words to Properties Using Python Dictionaries
● Study Rule based tagger, Unigram Tagger
Find different words from a given plain text without any space by comparing this text with a given corpus of words. Also find the score of words.

```
import nltk
from nltk.corpus import brown, inaugural, reuters, udhr
from nltk.corpus import stopwords
from nltk.probability import ConditionalFreqDist
from nltk.tag import UnigramTagger, pos_tag
from nltk.tokenize import word_tokenize
```

```
# Download necessary NLTK data files
nltk.download('brown')
nltk.download('inaugural')
nltk.download('reuters')
nltk.download('udhr')
nltk.download('stopwords')
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')
nltk.download('universal_tagset')

# 1. Study the various Corpora
print("Brown Corpus Categories:", brown.categories())
print("Inaugural Corpus Fileids:", inaugural.fileids())
print("Reuters Corpus Categories:", reuters.categories())
print("UDHR Corpus Languages:", udhr.languages())

# Access different parts of a corpus (e.g., raw text, words, sentences)
print("\nBrown Corpus - Raw Text Example:", brown.raw(categories='news')[:500])
print("\nBrown Corpus - Words Example:", brown.words(categories='news')[:10])
print("\nBrown Corpus - Sentences Example:", brown.sents(categories='news')[:2])

# 2. Create and use your own corpora (plaintext, categorical)
# Example of creating a simple text corpus
my_corpus = nltk.CorpusView.from_paths('my_corpus', ['sample_text.txt'])

# 3. Study Conditional Frequency Distributions
# Let's use the Brown Corpus for a conditional frequency distribution (tag vs. word)
tagged_brown = brown.tagged_words(categories='news')
cfreq = ConditionalFreqDist(
    (tag, word.lower()) for word, tag in tagged_brown
)
print("\nConditional Frequency Distribution for 'NN' tag:")
print(cfreq['NN'].most_common(10))  # Most common words with 'NN' tag

# 4. Study of Tagged Corpora
# Example: Getting tagged sentences from the Brown Corpus
tagged_sentences = brown.tagged_sents(categories='news')
print("\nTagged Sentences Example:", tagged_sentences[:2])

# Example: Finding the most frequent noun tags from a tagged corpus
tags = [tag for word, tag in tagged_brown]
freq_tags = nltk.FreqDist(tags)
print("\nMost Common Tags in Brown Corpus:", freq_tags.most_common(10))

# 5. Map Words to Properties Using Python Dictionaries
# Create a simple dictionary to map words to their frequencies
word_freq = nltk.FreqDist(brown.words(categories='news'))
print("\nFrequency of the word 'the':", word_freq['the'])

# 6. Rule-Based Tagger and Unigram Tagger
# Use a simple UnigramTagger with a portion of the Brown Corpus
train_sents = brown.tagged_sents(categories='news')[:3000]
```

```
test_sents = brown.tagged_sents(categories='news')[3000:3500]
unigram_tagger = UnigramTagger(train_sents)
tagged_test_sents = unigram_tagger.tag_sents([sent for sent in test_sents])

# Display the first few tagged sentences
print("\nTagged Sentences (Unigram Tagger) Example:")
print(tagged_test_sents[0])

# 7. Find words from a given plain text
# Assume a given plain text without spaces
text = "thisisaverysimpleexample"
words_from_corpus = set(brown.words(categories='news'))

# Find the words from the plain text
found_words = []
score = 0
start = 0
while start < len(text):
    for end in range(start + 1, len(text) + 1):
        word = text[start:end]
        if word in words_from_corpus:
            found_words.append(word)
            score += 1
            start = end
            break

print("\nWords found in the plain text:", found_words)
print("Total score (number of words found):", score)
```

OUTPUT:-

```
Brown Corpus Categories: ['adventure', 'belles_lettres', 'editorial', 'fiction',
'government', 'hobbies', 'humor', 'learned', 'lore', 'mystery', 'news',
'religion', 'reviews', 'romance', 'science_fiction']
Inaugural Corpus Fileids: ['1789-Washington.txt', '1793-Washington.txt', '1797-
Adams.txt', '1801-Jefferson.txt', '1805-Jefferson.txt', '1809-Madison.txt',
'1813-Madison.txt', '1817-Monroe.txt', '1821-Monroe.txt', '1825-Adams.txt',
'1829-Jackson.txt', '1833-Jackson.txt', '1837-VanBuren.txt', '1841-
Harrison.txt', '1845-Polk.txt', '1849-Taylor.txt', '1853-Pierce.txt', '1857-
Buchanan.txt', '1861-Lincoln.txt', '1865-Lincoln.txt', '1869-Grant.txt', '1873-
Grant.txt', '1877-Hayes.txt', '1881-Garfield.txt', '1885-Cleveland.txt', '1889-
Harrison.txt', '1893-Cleveland.txt', '1897-McKinley.txt', '1901-McKinley.txt',
'1905-Roosevelt.txt', '1909-Taft.txt', '1913-Wilson.txt', '1917-Wilson.txt',
'1921-Harding.txt', '1925-Coolidge.txt', '1929-Hoover.txt', '1933-
Roosevelt.txt', '1937-Roosevelt.txt', '1941-Roosevelt.txt', '1945-
Roosevelt.txt', '1949-Truman.txt', '1953-Eisenhower.txt', '1957-Eisenhower.txt',
'1961-Kennedy.txt', '1965-Johnson.txt', '1969-Nixon.txt', '1973-Nixon.txt',
'1977-Carter.txt', '1981-Reagan.txt', '1985-Reagan.txt', '1989-Bush.txt', '1993-
Clinton.txt', '1997-Clinton.txt', '2001-Bush.txt', '2005-Bush.txt', '2009-
Obama.txt', '2013-Obama.txt', '2017-Trump.txt', '2021-Biden.txt', '2025-
Trump.txt']
Reuters Corpus Categories: ['acq', 'alum', 'barley', 'bop', 'carcass', 'castor-
oil', 'cocoa', 'coconut', 'coconut-oil', 'coffee', 'copper', 'copra-cake',
'corn', 'cotton', 'cotton-oil', 'cpi', 'cpu', 'crude', 'dfl', 'dlr', 'dmk',
'earn', 'fuel', 'gas', 'gnp', 'gold', 'grain', 'groundnut', 'groundnut-oil',
'heat', 'hog', 'housing', 'income', 'instal-debt', 'interest', 'ipi', 'iron-
steel', 'jet', 'jobs', 'l-cattle', 'lead', 'lei', 'lin-oil', 'livestock',
'lumber', 'meal-feed', 'money-fx', 'money-supply', 'naphtha', 'nat-gas',
```

'nickel', 'nkr', 'nzdlr', 'oat', 'oilseed', 'orange', 'palladium', 'palm-oil', 'palmkernel', 'pet-chem', 'platinum', 'potato', 'propane', 'rand', 'rape-oil', 'rapeseed', 'reserves', 'retail', 'rice', 'rubber', 'rye', 'ship', 'silver', 'sorghum', 'soy-meal', 'soy-oil', 'soybean', 'strategic-metal', 'sugar', 'sun-meal', 'sun-oil', 'sunseed', 'tea', 'tin', 'trade', 'veg-oil', 'wheat', 'wpi', 'yen', 'zinc']

7.Write a Python program to find synonyms and antonyms of the word "active" using WordNet.

```
import nltk
from nltk.corpus import wordnet

# Download the WordNet data (run this once)
nltk.download('wordnet')

def find_synonyms_antonyms(word):
    # Get the synonyms and antonyms for the word
    synonyms = set()
    antonyms = set()

    # Get all synsets (synonym sets) of the word
    for syn in wordnet.synsets(word):
        # Add synonyms to the set
        for lemma in syn.lemmas():
            synonyms.add(lemma.name())

            # Check for antonyms
            if lemma.antonyms():
                antonyms.add(lemma.antonyms()[0].name())

    return synonyms, antonyms

# Test the function with the word "active"
word = "active"
synonyms, antonyms = find_synonyms_antonyms(word)

print(f"Synonyms of '{word}':", synonyms)
print(f"Antonyms of '{word}':", antonyms)
```

OUTPUT:-

```
Synonyms of 'active': {'participating', 'combat-ready', 'dynamic', 'fighting', 'alive', 'active', 'active_voice', 'active_agent'}
Antonyms of 'active': {'extinct', 'quiet', 'passive', 'passive_voice', 'dormant', 'stative', 'inactive'}
```

8.Implement the machine translation application of NLP where it needs to train a machine translation model for a language with limited parallel corpora. Investigate and incorporate techniques to improve performance in low-resource scenarios.

```
import tensorflow as tf
from tensorflow.keras import layers

# Toy data (English to Spanish)
source_text = ['hello', 'how are you', 'good morning']
target_text = ['hola', 'cómo estás', 'buenos días']

# Tokenize and pad sequences
source_tokenizer = tf.keras.preprocessing.text.Tokenizer()
target_tokenizer = tf.keras.preprocessing.text.Tokenizer()
source_tokenizer.fit_on_texts(source_text)
target_tokenizer.fit_on_texts(target_text)
source_padded =
tf.keras.preprocessing.sequence.pad_sequences(source_tokenizer.texts_to_sequences(source_text),
padding='post')
target_padded =
tf.keras.preprocessing.sequence.pad_sequences(target_tokenizer.texts_to_sequences(target_text),
padding='post')

# Define simple seq2seq model
encoder_inputs = layers.Input(shape=(None,))
encoder_embedding = layers.Embedding(input_dim=len(source_tokenizer.word_index) + 1,
output_dim=16)(encoder_inputs)
encoder_lstm = layers.LSTM(16, return_state=True)
encoder_outputs, state_h, state_c = encoder_lstm(encoder_embedding)

decoder_inputs = layers.Input(shape=(None,))
decoder_embedding = layers.Embedding(input_dim=len(target_tokenizer.word_index) + 1,
output_dim=16)(decoder_inputs)
decoder_lstm = layers.LSTM(16, return_sequences=True)(decoder_embedding,
initial_state=[state_h, state_c])
decoder_dense = layers.Dense(len(target_tokenizer.word_index) + 1, activation='softmax')
(decoder_lstm)

# Compile and train the model
model = tf.keras.models.Model([encoder_inputs, decoder_inputs], decoder_dense)
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
model.fit([source_padded, target_padded], target_padded, epochs=5)
```

OUTPUT;-

```
Epoch 1/5
1/1 ━━━━━━━━━━━━━━━━━━━━ 2s 2s/step - accuracy: 0.3333 - loss: 1.7919
Epoch 2/5
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - accuracy: 0.3333 - loss: 1.7902
Epoch 3/5
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 29ms/step - accuracy: 0.3333 - loss: 1.7884
Epoch 4/5
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 29ms/step - accuracy: 0.3333 - loss: 1.7866
Epoch 5/5
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 30ms/step - accuracy: 0.3333 - loss: 1.7849

<keras.src.callbacks.history.History at 0x7b29e4604490>
```