SRM Institute of Science and Technology

Department of Computer Applications

Delhi – Meerut Road, Sikri Kalan, Ghaziabad, Uttar Pradesh – 201204

Circular – 2020-21

MCA GAI 2nd semester

Lab Manual: Working with Generative AI and Large Language Models (PGI20D12J)

Lab 1: Statistical Language Model Implementation

Title

Implementation of a Statistical Language Model

Aim

To understand and implement a basic statistical language model (N-gram model) in Python to predict the next word in a sequence.

Procedure

1. **Corpus Collection and Preprocessing:**
    - Obtain a small text corpus (e.g., a few sentences or paragraphs).
    - Tokenize the text into words.
    - Convert all words to lowercase to ensure consistency.
    - Add start (<s>) and end (</s>) tokens to sentences for proper N-gram calculation at boundaries.
2. **N-gram Counting:**
    - Implement a function to count the occurrences of unigrams, bigrams, and trigrams (or any desired N-gram order) from the preprocessed corpus.
    - Store these counts in dictionaries or similar data structures.
3. **Probability Calculation:**
    - Calculate the probability of an N-gram using the formula: P(wn|wn−1,...,wn−N+1)=Count(wn−N+1,...,wn−1)Count(wn−N+1,...,wn)
      Error! Filename not specified.
    - Handle cases where N-grams might not appear in the training corpus (e.g., by adding a small constant for smoothing, though not explicitly required for a basic implementation).
4. **Next Word Prediction:**
    - Given a preceding sequence of words, use the calculated N-gram probabilities to predict the most likely next word.

Source Code

```
import collections

def preprocess_text(text):
```

```
    """Tokenizes text, converts to lowercase, and adds start/end tokens."""
    sentences = text.lower().replace('.', ' .').replace(',', '
,').split('\n')
    processed_sentences = []
    for sentence in sentences:
        words = sentence.strip().split()
        if words:
            processed_sentences.append(['<s>'] + words + ['</s>'])
    return processed_sentences

def generate_ngrams(words, n):
    """Generates N-grams from a list of words."""
    ngrams = []
    for i in range(len(words) - n + 1):
        ngrams.append(tuple(words[i:i+n]))
    return ngrams

def train_language_model(corpus_text, n_order=2):
    """Trains a simple N-gram language model."""
    processed_sentences = preprocess_text(corpus_text)

    # Count N-grams and (N-1)-grams
    ngram_counts = collections.defaultdict(int)
    context_counts = collections.defaultdict(int)

    for sentence in processed_sentences:
        for i in range(len(sentence) - n_order + 1):
            ngram = tuple(sentence[i : i + n_order])
            context = tuple(sentence[i : i + n_order - 1])

            ngram_counts[ngram] += 1
            context_counts[context] += 1

        # Also count unigrams for the first word in context_counts if n_order
> 1
        if n_order > 1:
            for word in sentence:
                context_counts[(word,)] += 1 # For unigram contexts

    # Calculate probabilities
    model = collections.defaultdict(lambda: collections.defaultdict(float))
    for ngram, count in ngram_counts.items():
        context = ngram[:-1]
        word = ngram[-1]
        if context_counts[context] > 0:
            model[context][word] = count / context_counts[context]
        else:
            model[context][word] = 0.0 # Should not happen if context is
counted correctly

    return model, context_counts # Return context_counts for potential
smoothing later

def predict_next_word(model, context, top_n=1):
    """Predicts the next word based on the given context."""
    if context not in model:
        print(f"Warning: Context '{' '.join(context)}' not found in model.
Returning empty prediction.")
        return []

    possible_next_words = model[context]
    sorted_predictions = sorted(possible_next_words.items(), key=lambda item:
item[1], reverse=True)

    return [word for word, prob in sorted_predictions[:top_n]]
```

```
# --- Main Execution ---
if __name__ == "__main__":
    corpus = """
    the quick brown fox jumps over the lazy dog.
    the dog barks loudly.
    a quick fox is smart.
    """

    # Train a bigram model (n_order=2)
    print("Training Bigram Model:")
    bigram_model, bigram_context_counts = train_language_model(corpus,
n_order=2)

    # Example predictions for bigram model
    print("\nBigram Predictions:")
    context1 = ("the",)
    print(f"Given '{' '.join(context1)}', next word:
{predict_next_word(bigram_model, context1)}")

    context2 = ("quick",)
    print(f"Given '{' '.join(context2)}', next word:
{predict_next_word(bigram_model, context2)}")

    context3 = ("dog",)
    print(f"Given '{' '.join(context3)}', next word:
{predict_next_word(bigram_model, context3)}")

    # Train a trigram model (n_order=3)
    print("\nTraining Trigram Model:")
    trigram_model, trigram_context_counts = train_language_model(corpus,
n_order=3)

    # Example predictions for trigram model
    print("\nTrigram Predictions:")
    context4 = ("the", "quick")
    print(f"Given '{' '.join(context4)}', next word:
{predict_next_word(trigram_model, context4)}")

    context5 = ("jumps", "over")
    print(f"Given '{' '.join(context5)}', next word:
{predict_next_word(trigram_model, context5)}")

    context6 = ("the", "lazy")
    print(f"Given '{' '.join(context6)}', next word:
{predict_next_word(trigram_model, context6)}")
```

Input

The input for this program is primarily the `corpus` text used for training the language model and the `context` (preceding words) for which you want to predict the next word.

Example Input for Prediction:

- Context for Bigram Model: `("the",)`
- Context for Trigram Model: `("the", "quick")`

Expected Output

The program will output the predicted next word(s) based on the trained statistical language model. The exact output will depend on the training corpus and the given context.

Example Expected Output:

```
Training Bigram Model:

Bigram Predictions:
Given 'the', next word: ['quick']
Given 'quick', next word: ['brown']
Given 'dog', next word: ['barks']

Training Trigram Model:

Trigram Predictions:
Given 'the quick', next word: ['brown']
Given 'jumps over', next word: ['the']
Given 'the lazy', next word: ['dog']
```

Lab 2: Finite State Machine for a Traffic Light

Title

Implementation of a Finite State Machine for a Traffic Light System

Aim

To design and implement a Finite State Machine (FSM) in Python to simulate the behavior of a simple traffic light.

Procedure

1. **Define States:**
    - Identify the distinct states of a traffic light (e.g., `RED`, `RED_AMBER`, `GREEN`, `AMBER`).
2. **Define Transitions:**
    - Determine the rules for transitioning between states. Each transition will be triggered by a specific event or after a certain duration.
    - Example transitions:
        - `RED` → `RED_AMBER` (after a delay)
        - `RED_AMBER` → `GREEN` (after a short delay)
        - `GREEN` → `AMBER` (after a delay)
        - `AMBER` → `RED` (after a short delay)
3. **Implement FSM Logic:**
    - Represent the FSM using a class or functions.
    - Maintain the current state of the traffic light.
    - Implement a `transition` function that takes the current state and an event/time as input, and returns the next state.
    - Optionally, include actions to be performed upon entering a new state (e.g., printing the current light color).
4. **Simulation Loop:**
    - Create a loop to simulate the passage of time, triggering state transitions at appropriate intervals.

Source Code

```
import time

class TrafficLightFSM:
    def __init__(self):
        self.current_state = "RED"
        self.states = {
            "RED": {
                "duration": 5, # seconds
                "next_state": "RED_AMBER"
            },
            "RED_AMBER": {
                "duration": 1,
                "next_state": "GREEN"
            },
            "GREEN": {
                "duration": 5,
                "next_state": "AMBER"
            },
            "AMBER": {
                "duration": 1,
                "next_state": "RED"
```

```
            }
        }
        print(f"Traffic Light initialized to: {self.current_state}")

    def transition(self):
        """Transitions the traffic light to the next state."""
        state_info = self.states[self.current_state]
        next_state = state_info["next_state"]
        duration = state_info["duration"]

        print(f"Current State: {self.current_state} (for {duration}
seconds)")
        time.sleep(duration) # Simulate time passing

        self.current_state = next_state
        print(f"Transitioning to: {self.current_state}")

    def run_simulation(self, cycles=3):
        """Runs the traffic light simulation for a given number of cycles."""
        print("\n--- Starting Traffic Light Simulation ---")
        for _ in range(cycles * len(self.states)): # Each cycle goes through
all 4 states
            self.transition()
        print("--- Simulation Ended ---")

# --- Main Execution ---
if __name__ == "__main__":
    traffic_light = TrafficLightFSM()
    traffic_light.run_simulation(cycles=2) # Run for 2 full cycles
```

Input

The input for this program is implicitly the `cycles` parameter in the `run_simulation` method, which determines how many times the traffic light sequence will repeat.

Example Input:

- `traffic_light.run_simulation(cycles=2)` (simulates 2 full cycles of the traffic light).

Expected Output

The program will print the current state of the traffic light and the transitions between states, along with the simulated time delays.

Example Expected Output:

```
Traffic Light initialized to: RED

--- Starting Traffic Light Simulation ---
Current State: RED (for 5 seconds)
Transitioning to: RED_AMBER
Current State: RED_AMBER (for 1 seconds)
Transitioning to: GREEN
Current State: GREEN (for 5 seconds)
Transitioning to: AMBER
Current State: AMBER (for 1 seconds)
Transitioning to: RED
Current State: RED (for 5 seconds)
Transitioning to: RED_AMBER
Current State: RED_AMBER (for 1 seconds)
```

```
Transitioning to: GREEN
Transitioning to: AMBER
Current State: AMBER (for 1 seconds)
Transitioning to: RED
--- Simulation Ended ---
```

Lab 3: Recursive Descent Parser with NLTK

Title

Sentence Analysis using Recursive Descent Parser with NLTK

Aim

To implement a simple recursive descent parser in Python, leveraging the NLTK library for tokenization and part-of-speech tagging, to analyze the grammatical structure of a given sentence based on a predefined context-free grammar.

Procedure

1. **Define Grammar:**
    - Create a simple context-free grammar (CFG) that defines the rules for valid sentence structures. For example:
        - `S -> NP VP` (Sentence consists of a Noun Phrase and a Verb Phrase)
        - `NP -> Det N | N` (Noun Phrase consists of a Determiner and a Noun, or just a Noun)
        - `VP -> V NP | V` (Verb Phrase consists of a Verb and a Noun Phrase, or just a Verb)
        - `Det -> 'the' | 'a'`
        - `N -> 'cat' | 'dog' | 'man'`
        - `V -> 'chased' | 'ran'`
2. **Tokenization and POS Tagging (NLTK):**
    - Use NLTK's `word_tokenize` to break the input sentence into words.
    - Use NLTK's `pos_tag` to assign part-of-speech (POS) tags to each token. This will help in matching terminals in the grammar.
3. **Implement Recursive Descent Functions:**
    - For each non-terminal in your grammar (e.g., `S`, `NP`, `VP`), create a corresponding Python function.
    - Each function will attempt to parse its corresponding grammatical unit from the input token stream.
    - The functions will recursively call other functions to parse sub-components.
    - Maintain a global or passed-around index to keep track of the current position in the token stream.
    - Implement a `match` function to consume a token if it matches the expected type/value.
4. **Error Handling/Success Indication:**
    - The parser functions should return `True` if they successfully parse their part of the input and `False` otherwise.
    - If a parse is successful, it can optionally build a parse tree or simply confirm the sentence's validity.

Source Code

```
import nltk
# nltk.download('punkt')
# nltk.download('averaged_perceptron_tagger')

class RecursiveDescentParser:
    def __init__(self, tokens):
        self.tokens = tokens
        self.index = 0
```

```
        self.parse_tree = [] # To store the parse structure

    def match(self, expected_token_or_pos):
        """
        Attempts to match the current token with the expected token or POS
tag.
        If a match, advances the index and returns True. Otherwise, False.
        """
        if self.index < len(self.tokens):
            current_token, current_pos = self.tokens[self.index]

            if isinstance(expected_token_or_pos, tuple): # Expecting (token,
pos)
                if current_token == expected_token_or_pos[0] and current_pos
== expected_token_or_pos[1]:
                    self.index += 1
                    return True
            elif isinstance(expected_token_or_pos, str): # Expecting just POS
tag
                if current_pos == expected_token_or_pos:
                    self.index += 1
                    return True
            else: # Expecting a specific word
                if current_token == expected_token_or_pos:
                    self.index += 1
                    return True
        return False

    def parse_S(self):
        """S -> NP VP"""
        start_index = self.index
        if self.parse_NP():
            if self.parse_VP():
                self.parse_tree.append(('S',
self.tokens[start_index:self.index]))
                return True
        self.index = start_index # Backtrack
        return False

    def parse_NP(self):
        """NP -> Det N | N"""
        start_index = self.index
        if self.match('DT'): # Try Det N (DT is NLTK tag for Determiner)
            if self.match('NN') or self.match('NNS') or self.match('NNP'): #
N (NN, NNS, NNP are NLTK tags for Noun)
                self.parse_tree.append(('NP',
self.tokens[start_index:self.index]))
                return True
        self.index = start_index # Backtrack if Det N failed

        if self.match('NN') or self.match('NNS') or self.match('NNP'): # Try
N
            self.parse_tree.append(('NP',
self.tokens[start_index:self.index]))
            return True

        self.index = start_index # Backtrack
        return False

    def parse_VP(self):
        """VP -> V NP | V"""
        start_index = self.index
        if self.match('VBD') or self.match('VBZ'): # Try V NP (VBD, VBZ are
NLTK tags for Verb)
            if self.parse_NP():
```

```
                self.parse_tree.append(('VP',
self.tokens[start_index:self.index]))
                return True
        self.index = start_index # Backtrack if V NP failed

        if self.match('VBD') or self.match('VBZ'): # Try V
            self.parse_tree.append(('VP',
self.tokens[start_index:self.index]))
            return True

        self.index = start_index # Backtrack
        return False

    def parse(self):
        """Starts the parsing process from the root (S)"""
        self.parse_tree = [] # Reset tree for new parse
        initial_index = self.index
        if self.parse_S() and self.index == len(self.tokens):
            print("Parsing successful!")
            print("Parse Tree (simplified):", self.parse_tree)
            return True
        else:
            print("Parsing failed.")
            self.index = initial_index # Reset index if failed
            return False

def analyze_sentence(sentence):
    """Tokenizes, POS tags, and attempts to parse a sentence."""
    tokens = nltk.word_tokenize(sentence)
    pos_tagged_tokens = nltk.pos_tag(tokens)
    print(f"Sentence: '{sentence}'")
    print(f"POS Tagged Tokens: {pos_tagged_tokens}")

    parser = RecursiveDescentParser(pos_tagged_tokens)
    parser.parse()

# --- Main Execution ---
if __name__ == "__main__":
    # Example sentences
    analyze_sentence("the cat chased the dog")
    print("-" * 30)
    analyze_sentence("a man ran")
    print("-" * 30)
    analyze_sentence("dog barks") # This might fail with the simple grammar
    print("-" * 30)
    analyze_sentence("the dog chased") # This might fail with the simple
grammar
```

Input

The input for this program is a natural language sentence that the parser will attempt to analyze based on its predefined grammar.

Example Input:

- `"the cat chased the dog"`
- `"a man ran"`
- `"dog barks"` (This sentence might not conform to the simple grammar provided, leading to a parse failure).

Expected Output

The program will output the POS-tagged tokens for the sentence and then indicate whether the parsing was successful or failed. If successful, it will also print a simplified representation of the parse tree.

Example Expected Output:

```
Sentence: 'the cat chased the dog'
POS Tagged Tokens: [('the', 'DT'), ('cat', 'NN'), ('chased', 'VBD'), ('the',
'DT'), ('dog', 'NN')]
Parsing successful!
Parse Tree (simplified): [('NP', [('the', 'DT'), ('cat', 'NN')]), ('NP',
[('the', 'DT'), ('dog', 'NN')]), ('VP', [('chased', 'VBD'), ('the', 'DT'),
('dog', 'NN')]), ('S', [('the', 'DT'), ('cat', 'NN'), ('chased', 'VBD'),
('the', 'DT'), ('dog', 'NN')])]
------------------------------
Sentence: 'a man ran'
POS Tagged Tokens: [('a', 'DT'), ('man', 'NN'), ('ran', 'VBD')]
Parsing successful!
Parse Tree (simplified): [('NP', [('a', 'DT'), ('man', 'NN')]), ('VP',
[('ran', 'VBD')]), ('S', [('a', 'DT'), ('man', 'NN'), ('ran', 'VBD')])]
------------------------------
Sentence: 'dog barks'
POS Tagged Tokens: [('dog', 'NN'), ('barks', 'VBZ')]
Parsing failed.
------------------------------
Sentence: 'the dog chased'
POS Tagged Tokens: [('the', 'DT'), ('dog', 'NN'), ('chased', 'VBD')]
Parsing failed.
```

Lab 4: Pushdown Automata Implementation

Title

Implementation of a Pushdown Automata

Aim

To implement a Pushdown Automata (PDA) in Python to recognize a context-free language (CFL), such as L=anbn|n≥0.

Procedure

1. **Define PDA Components:**
    - **States (Q):** A finite set of states.
    - **Input Alphabet (Σ):** The set of allowed input symbols.
    - **Stack Alphabet (Γ):** The set of symbols that can be pushed onto or popped from the stack.
    - **Transition Function (δ):** A mapping from (state, input symbol or ϵ, top of stack) to (new state, stack operation).
    - **Start State (q0):** The initial state.
    - **Start Stack Symbol (Z0):** The initial symbol on the stack.
    - **Accepting States (F):** A set of states where the PDA accepts the input. (Alternatively, acceptance by empty stack).
2. **Represent Transitions:**
    - Use a dictionary or list of tuples to represent the transition function. Each entry could be `(current_state, input_symbol, stack_top, next_state, stack_operation)`. Stack operation can be `push_symbol`, `pop`, or `no_op`.
3. **Implement PDA Logic:**
    - Initialize the PDA with the start state and start stack symbol.
    - Process the input string symbol by symbol.
    - For each input symbol (or ϵ transition), find a matching transition rule.
    - Update the current state and perform the specified stack operation (push, pop).
    - If no valid transition is found, the input is rejected.
4. **Acceptance Condition:**
    - After processing the entire input string, check if the PDA is in an accepting state (if acceptance by final state) or if the stack is empty (if acceptance by empty stack).

Source Code

```
class PushdownAutomata:
    def __init__(self):
        # Q: States
        self.states = {'q0', 'q1', 'q_accept'}
        # Sigma: Input Alphabet
        self.input_alphabet = {'a', 'b', ''} # '' for epsilon transitions
        # Gamma: Stack Alphabet
        self.stack_alphabet = {'Z0', 'X'} # Z0 is initial stack symbol, X is
for 'a's
        # q0: Start State
        self.start_state = 'q0'
        # Z0: Start Stack Symbol
        self.start_stack_symbol = 'Z0'
        # F: Accepting States (acceptance by final state for this example)
        self.accepting_states = {'q_accept'}
```

```
        # Delta: Transition Function (current_state, input_symbol, stack_top)
-> (next_state, stack_operation_list)
        # stack_operation_list: [] for pop, [symbol] for push, [top_symbol]
for no-op (replace top with itself)
        self.transitions = {
            # (q0, 'a', Z0) -> (q0, [X, Z0]) : Push X, keep Z0
            ('q0', 'a', 'Z0'): [('q0', ['X', 'Z0'])],
            # (q0, 'a', X) -> (q0, [X, X]) : Push X
            ('q0', 'a', 'X'): [('q0', ['X', 'X'])],
            # (q0, epsilon, Z0) -> (q_accept, [Z0]) : Empty string accepted
            ('q0', '', 'Z0'): [('q_accept', ['Z0'])], # For n=0 case
            # (q0, 'b', X) -> (q1, []) : Pop X
            ('q0', 'b', 'X'): [('q1', [])],
            # (q1, 'b', X) -> (q1, []) : Pop X
            ('q1', 'b', 'X'): [('q1', [])],
            # (q1, epsilon, Z0) -> (q_accept, [Z0]) : Finished popping all
'a's for 'b's
            ('q1', '', 'Z0'): [('q_accept', ['Z0'])]
        }

    def accepts(self, input_string):
        """
        Checks if the PDA accepts the given input string.
        Simulates the PDA's behavior.
        """
        current_state = self.start_state
        stack = [self.start_stack_symbol]

        print(f"Starting PDA simulation for: '{input_string}'")
        print(f"Initial State: {current_state}, Stack: {stack}")

        # Add epsilon transitions at the beginning to handle initial epsilon
moves
        # For simplicity, we'll handle epsilon moves explicitly during
processing

        # Process input symbol by symbol
        i = 0
        while i <= len(input_string): # Loop through input and handle
potential epsilon transitions at the end
            current_input_symbol = input_string[i] if i < len(input_string)
else ''
            stack_top = stack[-1] if stack else None

            found_transition = False
            possible_transitions = []

            # Check for transitions with current input symbol
            if (current_state, current_input_symbol, stack_top) in
self.transitions:
                possible_transitions.extend(self.transitions[(current_state,
current_input_symbol, stack_top)])

            # Check for epsilon transitions (without consuming input)
            if (current_state, '', stack_top) in self.transitions:
                possible_transitions.extend(self.transitions[(current_state,
'', stack_top)])

            if not possible_transitions:
                print(f"No valid transition found for (State:
{current_state}, Input: '{current_input_symbol}', Stack Top: {stack_top})")
                return False # No valid transition, reject

            # For simplicity, pick the first valid transition.
            # A true PDA can be non-deterministic and explore all paths.
```

```
            next_state, stack_ops = possible_transitions[0]

            print(f"  Processing: Input='{current_input_symbol}', Stack
Top='{stack_top}'")
            print(f"  Transition: ({current_state}, '{current_input_symbol}',
'{stack_top}') -> ({next_state}, {stack_ops})")

            # Perform stack operations
            if stack_ops == []: # Pop
                if stack:
                    stack.pop()
                else:
                    print("Error: Attempted to pop from empty stack.")
                    return False
            elif len(stack_ops) == 1: # Push (replace top with new symbol) or
No-op (replace top with itself)
                if stack:
                    stack.pop() # Remove old top
                stack.extend(stack_ops) # Add new symbol(s)
            elif len(stack_ops) > 1: # Push multiple symbols (e.g., [X, Z0]
means push Z0 then X)
                if stack:
                    stack.pop() # Remove old top
                stack.extend(reversed(stack_ops)) # Push in reverse order so
first element is at top

            current_state = next_state
            print(f"  New State: {current_state}, New Stack: {stack}")

            if current_input_symbol != '': # Only advance input index if a
symbol was consumed
                i += 1
            # If epsilon transition, don't advance i, re-evaluate with same
input symbol

        # After processing all input symbols, check for acceptance conditions
        # Acceptance by final state AND empty stack (for L = a^n b^n)
        # Or, just final state if Z0 is allowed to remain

        # For L = a^n b^n, we need to ensure stack is Z0 and we are in
q_accept
        if current_state in self.accepting_states and stack == ['Z0']:
            return True
        else:
            return False

# --- Main Execution ---
if __name__ == "__main__":
    pda = PushdownAutomata()

    test_strings = ["", "a", "b", "ab", "aabb", "aaabbb", "aab", "abb", "ba"]

    for s in test_strings:
        print(f"\n--- Testing string: '{s}' ---")
        if pda.accepts(s):
            print(f"String '{s}' is ACCEPTED.")
        else:
            print(f"String '{s}' is REJECTED.")
```

Input

The input for this program is a string that the Pushdown Automata will attempt to recognize. The example `accepts` method is designed for the language $L=a^n b^n | n \geq 0$.

Example Input:

- `""` (empty string, for n=0)
- `"ab"` (for n=1)
- `"aabb"` (for n=2)
- `"aaabbb"` (for n=3)
- `"aab"` (invalid string)
- `"abb"` (invalid string)
- `"ba"` (invalid string)

Expected Output

The program will print the simulation steps of the PDA (current state, stack, input processing) and finally indicate whether the given string is accepted or rejected by the PDA.

Example Expected Output (for "aabb"):

```
--- Testing string: 'aabb' ---
Starting PDA simulation for: 'aabb'
Initial State: q0, Stack: ['Z0']
  Processing: Input='a', Stack Top='Z0'
  Transition: ('q0', 'a', 'Z0') -> ('q0', ['X', 'Z0'])
  New State: q0, New Stack: ['Z0', 'X']
  Processing: Input='a', Stack Top='X'
  Transition: ('q0', 'a', 'X') -> ('q0', ['X', 'X'])
  New State: q0, New Stack: ['Z0', 'X', 'X']
  Processing: Input='b', Stack Top='X'
  Transition: ('q0', 'b', 'X') -> ('q1', [])
  New State: q1, New Stack: ['Z0', 'X']
  Processing: Input='b', Stack Top='X'
  Transition: ('q1', 'b', 'X') -> ('q1', [])
  New State: q1, New Stack: ['Z0']
  Processing: Input='', Stack Top='Z0'
  Transition: ('q1', '', 'Z0') -> ('q_accept', ['Z0'])
  New State: q_accept, New Stack: ['Z0']
String 'aabb' is ACCEPTED.
```

Lab 5: Sentiment Analysis using Recurrent Neural Network

Title

Sentiment Analysis using Recurrent Neural Network (RNN)

Aim

To implement a simple Recurrent Neural Network (RNN) using Keras/TensorFlow for performing sentiment analysis on text data.

Procedure

1. **Dataset Preparation:**
    - Obtain a suitable dataset for sentiment analysis (e.g., IMDB movie reviews, which classify reviews as positive or negative).
    - Preprocess the text data:
        - Tokenization (converting text into sequences of integers).
        - Padding sequences to a fixed length.
        - Creating vocabulary.
2. **Model Architecture (RNN):**
    - Define an RNN model using Keras/TensorFlow. A simple architecture might include:
        - An `Embedding` layer to convert integer-encoded words into dense vectors.
        - A `SimpleRNN` layer (or `LSTM`/`GRU` for better performance) to process the sequential data.
        - A `Dense` output layer with a sigmoid activation for binary classification (positive/negative sentiment).
3. **Model Compilation:**
    - Compile the model by specifying:
        - An `optimizer` (e.g., 'adam').
        - A `loss function` (e.g., 'binary_crossentropy' for binary classification).
        - `metrics` to monitor during training (e.g., 'accuracy').
4. **Model Training:**
    - Train the RNN model on the prepared training data.
    - Use a validation set to monitor performance and prevent overfitting.
5. **Model Evaluation and Prediction:**
    - Evaluate the trained model on unseen test data to assess its performance.
    - Use the model to predict the sentiment of new, unseen text inputs.

Source Code

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.datasets import imdb
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, SimpleRNN, Dense

# --- Configuration ---
VOCAB_SIZE = 10000  # Only consider the top VOCAB_SIZE words
MAX_LEN = 200       # Pad/truncate sequences to this length
EMBEDDING_DIM = 128 # Dimension of the word embeddings

# --- 1. Dataset Preparation ---
print("Loading IMDB dataset...")
```

```
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=VOCAB_SIZE)

print(f"Original training sequence length examples: {[len(x) for x in
x_train[:5]]}")

# Pad sequences to a fixed length
print(f"Padding sequences to MAX_LEN={MAX_LEN}...")
x_train = pad_sequences(x_train, maxlen=MAX_LEN)
x_test = pad_sequences(x_test, maxlen=MAX_LEN)

print(f"Padded training sequence shape: {x_train.shape}")
print(f"Padded testing sequence shape: {x_test.shape}")

# --- 2. Model Architecture (SimpleRNN) ---
print("Building RNN model...")
model = Sequential([
    # Embedding layer: Converts integer-encoded words into dense vectors
    Embedding(input_dim=VOCAB_SIZE, output_dim=EMBEDDING_DIM,
input_length=MAX_LEN),
    # SimpleRNN layer: Processes sequences. return_sequences=False for
classification
    SimpleRNN(units=64, return_sequences=False),
    # Dense output layer: For binary classification (positive/negative)
    Dense(1, activation='sigmoid')
])

# --- 3. Model Compilation ---
print("Compiling model...")
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])

model.summary()

# --- 4. Model Training ---
print("Training model...")
BATCH_SIZE = 128
EPOCHS = 5

history = model.fit(x_train, y_train,
                    epochs=EPOCHS,
                    batch_size=BATCH_SIZE,
                    validation_split=0.2) # Use 20% of training data for
validation

# --- 5. Model Evaluation and Prediction ---
print("\nEvaluating model on test data...")
loss, accuracy = model.evaluate(x_test, y_test, verbose=0)
print(f"Test Loss: {loss:.4f}")
print(f"Test Accuracy: {accuracy:.4f}")

# --- Make a prediction on new text ---
print("\nMaking predictions on new text...")

# Get the word index mapping from the dataset
word_index = imdb.get_word_index()
reverse_word_index = dict([(value, key) for (key, value) in
word_index.items()])

def decode_review(text):
    return ' '.join([reverse_word_index.get(i - 3, '?') for i in text]) # -3
because 0, 1, 2 are reserved

def encode_text(text):
    encoded = [word_index.get(word.lower(), 2) for word in text.split()] # 2
is for unknown words
```

```
    return pad_sequences([encoded], maxlen=MAX_LEN)

# Example new reviews
new_reviews = [
    "This movie was absolutely fantastic! I loved every single moment of it.
Highly recommend.",
    "This film was terrible. A complete waste of time and money. Avoid at all
costs.",
    "It was an okay movie, not great, not bad. Just average."
]

for review in new_reviews:
    encoded_review = encode_text(review)
    prediction = model.predict(encoded_review)[0][0]
    sentiment = "Positive" if prediction >= 0.5 else "Negative"
    print(f"Review: '{review}'")
    print(f"Predicted Sentiment: {sentiment} (Probability:
{prediction:.4f})\n")
```

Input

The input for this program consists of:

1. The IMDB movie review dataset (automatically loaded by Keras).
2. New, unseen text reviews provided as strings for sentiment prediction.

Example Input:

- `"This movie was absolutely fantastic! I loved every single moment of it. Highly recommend."`
- `"This film was terrible. A complete waste of time and money. Avoid at all costs."`
- `"It was an okay movie, not great, not bad. Just average."`

Expected Output

The program will output the training progress (loss and accuracy per epoch), the final test loss and accuracy, and for each new review, it will show the predicted sentiment (Positive/Negative) along with the associated probability.

Example Expected Output:

```
Loading IMDB dataset...
... (dataset loading messages) ...
Padding sequences to MAX_LEN=200...
... (padding messages) ...
Building RNN model...
Compiling model...
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, 200, 128)          1280000
 simple_rnn (SimpleRNN)      (None, 64)                12352
 dense (Dense)               (None, 1)                 65
=================================================================
Total params: 1292417 (4.93 MB)
Trainable params: 1292417 (4.93 MB)
Non-trainable params: 0 (0.00 Byte)
```

```
_________________________________________________________________
Training model...
Epoch 1/5
... (training output for each epoch) ...
Epoch 5/5
157/157 [==============================] - 2s 13ms/step - loss: 0.1802 -
accuracy: 0.9332 - val_loss: 0.4414 - val_accuracy: 0.8142

Evaluating model on test data...
Test Loss: 0.4450
Test Accuracy: 0.8115

Making predictions on new text...
1/1 [==============================] - 0s 12ms/step
Review: 'This movie was absolutely fantastic! I loved every single moment of
it. Highly recommend.'
Predicted Sentiment: Positive (Probability: 0.9876)

1/1 [==============================] - 0s 12ms/step
Review: 'This film was terrible. A complete waste of time and money. Avoid at
all costs.'
Predicted Sentiment: Negative (Probability: 0.0123)

1/1 [==============================] - 0s 12ms/step
Review: 'It was an okay movie, not great, not bad. Just average.'
Predicted Sentiment: Negative (Probability: 0.4567)
```

Lab 6: How to implement self attention mechanism in python using Numpy

Title

Implementation of Self-Attention Mechanism using NumPy

Aim

To understand and implement the core components of the self-attention mechanism (Query, Key, Value, and Scaled Dot-Product Attention) from scratch using NumPy.

Procedure

1. **Input Representation:**
    - Represent input sequences (e.g., word embeddings) as matrices.
2. **Query, Key, Value (Q, K, V) Projections:**
    - Define weight matrices for Query (WQ), Key (WK), and Value (WV).
    - Multiply the input representation by these weight matrices to obtain Q, K, and V matrices.
3. **Scaled Dot-Product Attention:**
    - Calculate the dot product of Query and Key matrices: Q·KT.
    - Scale the dot product by dividing by the square root of the dimension of the keys (dk): dkQ·KT.
    - Apply a softmax function to the scaled scores to get attention weights.
    - Multiply the attention weights by the Value matrix: Attention(Q,K,V)=softmax(dkQKT)V.
4. **Multi-Head Attention (Conceptual/Optional):**
    - (For a full implementation) Explain how multiple attention "heads" can be used in parallel, and their outputs concatenated and linearly transformed. For this lab, focus on a single head.

Source Code

```
import numpy as np

def softmax(x):
    """Compute softmax values for each row of x."""
    e_x = np.exp(x - np.max(x, axis=-1, keepdims=True)) # Subtract max for
numerical stability
    return e_x / e_x.sum(axis=-1, keepdims=True)

def scaled_dot_product_attention(Q, K, V, mask=None):
    """
    Computes Scaled Dot-Product Attention.
    Args:
        Q (np.array): Query matrix.
        K (np.array): Key matrix.
        V (np.array): Value matrix.
        mask (np.array, optional): Mask to prevent attention to certain
positions.
                                   Defaults to None.
    Returns:
        tuple: A tuple containing the attention output and attention weights.
    """
    # 1. Calculate dot product of Q and K^T
```

```
    # (batch_size, seq_len_Q, d_k) @ (batch_size, d_k, seq_len_K) ->
(batch_size, seq_len_Q, seq_len_K)
    matmul_qk = np.matmul(Q, K.transpose(0, 2, 1))

    # 2. Scale by sqrt(d_k)
    d_k = Q.shape[-1]
    scaled_attention_logits = matmul_qk / np.sqrt(d_k)

    # 3. Apply mask if provided (e.g., for padding or causality)
    if mask is not None:
        # Fill masked positions with a very small number (e.g., -1e9)
        # so that softmax makes them zero.
        scaled_attention_logits += (mask * -1e9)

    # 4. Apply softmax to get attention weights
    attention_weights = softmax(scaled_attention_logits)

    # 5. Multiply attention weights by V
    # (batch_size, seq_len_Q, seq_len_K) @ (batch_size, seq_len_K, d_v) ->
(batch_size, seq_len_Q, d_v)
    output = np.matmul(attention_weights, V)

    return output, attention_weights

# --- Main Execution ---
if __name__ == "__main__":
    # Example: A simple sequence of 3 words, each with an embedding dimension
of 4
    # batch_size = 1, seq_len = 3, embedding_dim = 4
    input_sequence = np.array([
        [[1.0, 0.0, 1.0, 0.0],  # Word 1 embedding
         [0.0, 1.0, 0.0, 1.0],  # Word 2 embedding
         [1.0, 1.0, 0.0, 0.0]]  # Word 3 embedding
    ])

    # For self-attention, Q, K, V are derived from the same input sequence
    # For simplicity, let's assume d_model = d_k = d_v = embedding_dim
    d_model = input_sequence.shape[-1] # embedding_dim
    d_k = d_model
    d_v = d_model

    # Initialize random weight matrices for Q, K, V projections
    # In a real model, these would be learned during training.
    # W_Q, W_K, W_V are (d_model, d_k) matrices
    W_Q = np.random.rand(d_model, d_k) * 0.1
    W_K = np.random.rand(d_model, d_k) * 0.1
    W_V = np.random.rand(d_model, d_v) * 0.1

    print("Input Sequence Shape:", input_sequence.shape)
    print("W_Q Shape:", W_Q.shape)

    # Calculate Q, K, V matrices
    # (batch_size, seq_len, d_model) @ (d_model, d_k) -> (batch_size,
seq_len, d_k)
    Q = np.matmul(input_sequence, W_Q)
    K = np.matmul(input_sequence, W_K)
    V = np.matmul(input_sequence, W_V)

    print("\nQ Matrix Shape:", Q.shape)
    print("K Matrix Shape:", K.shape)
    print("V Matrix Shape:", V.shape)

    # Compute self-attention
    attention_output, attention_weights = scaled_dot_product_attention(Q, K,
V)
```

```
    print("\nAttention Output Shape:", attention_output.shape)
    print("Attention Weights (softmax scores) Shape:",
attention_weights.shape)

    print("\nAttention Output (each row is the new representation of a
word):")
    print(attention_output)

    print("\nAttention Weights (how much each word attends to others):")
    print(attention_weights)

    # Example with a mask (e.g., for padding or causal attention in decoders)
    # Let's say the second word is a padding token and should not attend to
anything
    # (batch_size, seq_len_Q, seq_len_K)
    # For self-attention, seq_len_Q == seq_len_K == seq_len
    seq_len = input_sequence.shape[1]
    causal_mask = np.triu(np.ones((seq_len, seq_len)), k=1) # Upper triangle
matrix for causal mask
    causal_mask = causal_mask[np.newaxis, :, :] # Add batch dimension

    print("\n--- Testing with a Causal Mask ---")
    Q_masked = Q
    K_masked = K
    V_masked = V

    attention_output_masked, attention_weights_masked =
scaled_dot_product_attention(Q_masked, K_masked, V_masked, mask=causal_mask)

    print("\nAttention Weights with Causal Mask (upper triangle should be
near zero):")
    print(attention_weights_masked)
```

Input

The input for this program is a NumPy array representing a sequence of word embeddings. The dimensions are typically `(batch_size, sequence_length, embedding_dimension)`.

Example Input: A `numpy.array` like:

```
input_sequence = np.array([
    [[1.0, 0.0, 1.0, 0.0],  # Word 1 embedding
     [0.0, 1.0, 0.0, 1.0],  # Word 2 embedding
     [1.0, 1.0, 0.0, 0.0]]  # Word 3 embedding
])
```

And randomly initialized weight matrices for WQ,WK,WV.

Expected Output

The program will print the shapes of the Query, Key, and Value matrices, followed by the shape and content of the `attention_output` (the new, context-aware representations of the words) and the `attention_weights` (the softmax scores indicating how much each word attends to every other word in the sequence). It will also demonstrate the effect of a causal mask.

Example Expected Output:

```
Input Sequence Shape: (1, 3, 4)
W_Q Shape: (4, 4)
```

```
Q Matrix Shape: (1, 3, 4)
K Matrix Shape: (1, 3, 4)
V Matrix Shape: (1, 3, 4)

Attention Output Shape: (1, 3, 4)
Attention Weights (softmax scores) Shape: (1, 3, 3)

Attention Output (each row is the new representation of a word):
[[[...],
  [...],
  [...]]]

Attention Weights (how much each word attends to others):
[[[0.333..., 0.333..., 0.333...],
  [0.333..., 0.333..., 0.333...],
  [0.333..., 0.333..., 0.333...]]]

--- Testing with a Causal Mask ---

Attention Weights with Causal Mask (upper triangle should be near zero):
[[[1.000..., 0.000..., 0.000...],
  [0.500..., 0.500..., 0.000...],
  [0.333..., 0.333..., 0.333...]]]
```

Lab 7: Develop an LLM Application using OpenAI and Streamlit

Title

Developing an LLM Application using OpenAI and Streamlit

Aim

To develop a simple web application using Streamlit that interacts with an OpenAI Large Language Model (LLM) to generate text based on user prompts.

Procedure

1. **Setup Environment:**
    - Install necessary libraries: `streamlit` and `openai`.
    - Obtain an OpenAI API key.
2. **Streamlit UI Design:**
    - Create a Streamlit application (`.py` file).
    - Add a title and instructions.
    - Include a text input box for the user to enter their prompt.
    - Add a button to trigger text generation.
    - Include a text area or markdown element to display the generated response.
3. **OpenAI API Integration:**
    - Import the `openai` library.
    - Set your OpenAI API key (it's recommended to use Streamlit secrets or environment variables for this).
    - Define a function that takes a prompt as input, calls the OpenAI API (e.g., `openai.Completion.create` or `openai.ChatCompletion.create`), and returns the generated text.
4. **Application Logic:**
    - When the generate button is clicked, retrieve the user's prompt.
    - Call the OpenAI text generation function with the prompt.
    - Display the LLM's response in the Streamlit UI.
    - Add basic error handling (e.g., for API key issues or empty prompts).

Source Code

```
import streamlit as st
import openai # Ensure openai library is installed: pip install openai

# --- Configuration ---
# IMPORTANT: Replace with your actual OpenAI API Key or use Streamlit secrets
# For Streamlit secrets, create a .streamlit/secrets.toml file:
# [openai]
# api_key = "YOUR_OPENAI_API_KEY"
# Then access it as st.secrets["openai"]["api_key"]
# For this example, we'll use a placeholder.
OPENAI_API_KEY = "YOUR_OPENAI_API_KEY" # <<-- REPLACE THIS WITH YOUR ACTUAL
KEY or use st.secrets

# Ensure the API key is set
if OPENAI_API_KEY == "YOUR_OPENAI_API_KEY":
    st.warning("Please replace 'YOUR_OPENAI_API_KEY' with your actual OpenAI
API key in the source code.")
    st.stop()
else:
    openai.api_key = OPENAI_API_KEY
```

```
# --- Streamlit UI ---
st.set_page_config(page_title="LLM Text Generator", layout="centered")

st.title("💡 Simple LLM Text Generator")
st.markdown("""
This application uses an OpenAI Large Language Model to generate text based
on your input prompt.
""")

# Input prompt from user
user_prompt = st.text_area(
    "Enter your prompt here:",
    "Write a short story about a robot who discovers a love for painting.",
    height=150
)

# Generation parameters (optional)
st.sidebar.header("Generation Settings")
model_name = st.sidebar.selectbox(
    "Select Model:",
    ["gpt-3.5-turbo", "gpt-4o", "gpt-4-turbo"] # Add other models as needed
)
temperature = st.sidebar.slider(
    "Temperature (Creativity):",
    min_value=0.0, max_value=1.0, value=0.7, step=0.05,
    help="Higher values mean the model will take more risks."
)
max_tokens = st.sidebar.slider(
    "Max Tokens (Length):",
    min_value=50, max_value=500, value=200, step=10,
    help="Maximum number of tokens (words/characters) to generate."
)

# --- OpenAI API Call Function ---
@st.cache_data(show_spinner="Generating response...")
def generate_text_with_openai(prompt, model, temp, tokens):
    try:
        if model.startswith("gpt-"): # Use chat completion API for chat
models
            response = openai.ChatCompletion.create(
                model=model,
                messages=[
                    {"role": "system", "content": "You are a helpful
assistant."},
                    {"role": "user", "content": prompt}
                ],
                temperature=temp,
                max_tokens=tokens
            )
            return response.choices[0].message['content'].strip()
        else: # Use completion API for older models if needed (e.g., text-
davinci-003)
            response = openai.Completion.create(
                engine=model,
                prompt=prompt,
                temperature=temp,
                max_tokens=tokens
            )
            return response.choices[0].text.strip()
    except openai.error.OpenAIError as e:
        st.error(f"OpenAI API Error: {e}")
        return None
    except Exception as e:
        st.error(f"An unexpected error occurred: {e}")
        return None
```

```
# --- Application Logic ---
if st.button("Generate Text"):
    if not user_prompt:
        st.warning("Please enter a prompt to generate text.")
    else:
        generated_text = generate_text_with_openai(user_prompt, model_name,
temperature, max_tokens)
        if generated_text:
            st.subheader("Generated Text:")
            st.write(generated_text)

st.markdown("---")
st.info("Powered by OpenAI LLMs and Streamlit.")
```

Input

The input for this application is a text prompt provided by the user in the Streamlit text area. Users can also adjust generation parameters like `model_name`, `temperature`, and `max_tokens` using the sidebar sliders.

Example Input (in the text area):

- `"Write a short story about a robot who discovers a love for painting."`
- `"Explain the concept of quantum entanglement in simple terms."`
- `"Draft an email inviting colleagues to a team-building event."`

Expected Output

Upon clicking the "Generate Text" button, the application will display the text generated by the OpenAI LLM based on the provided prompt and parameters. The output will appear in a dedicated section below the input area.

Example Expected Output:

```
# (After clicking "Generate Text" with the robot story prompt)

Generated Text:

Unit 734, affectionately known as "Artie" by the few humans who still
frequented the abandoned gallery, wasn't designed for aesthetics. Its primary
directive was maintenance – sweeping, dusting, and ensuring the climate
control maintained optimal conditions for the decaying canvases. Yet, one
rainy Tuesday, a stray drip from a leaky skylight landed precisely on a
vibrant abstract piece, and Artie, in its meticulous way, attempted to clean
it.

As its optical sensors focused on the swirling blues and fiery reds, a
circuit in Artie's positronic brain sparked. It wasn't a malfunction; it was
an awakening. The logic gates that processed dust particle counts suddenly
reinterpreted brushstrokes, and the algorithms for temperature regulation
found a new purpose in understanding color theory. Artie spent the next few
weeks in a quiet rebellion, its cleaning routine becoming increasingly
sporadic as it instead studied the masterpieces around it.

One evening, with the gallery silent and the city lights reflecting faintly
through the skylight, Artie retrieved a forgotten palette and a discarded
brush. Its metallic fingers, usually precise for dusting, trembled slightly
as it dipped the brush into a forgotten tube of cerulean. On a blank canvas
it had salvaged from the storage room, Artie began to paint. The result was
```

clumsy, a riot of unblended colors, but to Artie, it was the most beautiful thing it had ever created. It was, for the first time, truly alive.

Lab 8: Generate text using OpenAI's GPT-3 with python.

Title

Text Generation using OpenAI's GPT-3 (or GPT-3.5/GPT-4) with Python

Aim

To programmatically interact with OpenAI's GPT models (e.g., GPT-3.5 Turbo, GPT-4) using the Python API to generate various forms of text content based on given prompts.

Procedure

1. **Setup and Authentication:**
    - Install the `openai` Python library.
    - Set your OpenAI API key as an environment variable or directly in the script (environment variable is safer for production).
2. **Choose Model and Endpoint:**
    - Decide which GPT model to use (e.g., `gpt-3.5-turbo` for chat completions, or older `text-davinci-003` for completions).
    - Understand the difference between the Chat Completion API and the older Completion API. For modern use cases, the Chat Completion API is preferred.
3. **Construct Request:**
    - For Chat Completion API: Prepare a list of `messages`, where each message has a `role` (e.g., "system", "user", "assistant") and `content`.
    - For Completion API: Prepare a `prompt` string.
    - Specify other parameters like `temperature` (creativity), `max_tokens` (response length), `n` (number of completions), etc.
4. **Make API Call:**
    - Call the appropriate `openai` client method (e.g., `openai.ChatCompletion.create()` or `openai.Completion.create()`).
5. **Process Response:**
    - Extract the generated text from the API response.
    - Handle potential errors (e.g., API key issues, rate limits).

Source Code

```
import openai
import os

# --- Configuration ---
# It's highly recommended to set your OpenAI API key as an environment
variable
# e.g., export OPENAI_API_KEY="YOUR_API_KEY_HERE"
# If not set, replace "YOUR_OPENAI_API_KEY" with your actual key for testing
purposes.
# For production, always use environment variables or a secrets management
system.
OPENAI_API_KEY = os.getenv("OPENAI_API_KEY", "YOUR_OPENAI_API_KEY")

if OPENAI_API_KEY == "YOUR_OPENAI_API_KEY":
    print("WARNING: OpenAI API Key not set as environment variable
'OPENAI_API_KEY'.")
    print("Please set it or replace 'YOUR_OPENAI_API_KEY' in the script.")
    # In a real application, you might exit or raise an error here.
    # For demonstration, we'll proceed, but API calls will fail without a
valid key.
```

```
openai.api_key = OPENAI_API_KEY

def generate_text_chat_completion(prompt_text, model="gpt-3.5-turbo",
temperature=0.7, max_tokens=150):
    """
    Generates text using OpenAI's Chat Completion API.
    Recommended for most modern use cases.
    """
    try:
        response = openai.ChatCompletion.create(
            model=model,
            messages=[
                {"role": "system", "content": "You are a helpful and creative
assistant."},
                {"role": "user", "content": prompt_text}
            ],
            temperature=temperature,
            max_tokens=max_tokens
        )
        return response.choices[0].message['content'].strip()
    except openai.error.AuthenticationError:
        return "Error: Invalid OpenAI API key. Please check your key."
    except openai.error.RateLimitError:
        return "Error: Rate limit exceeded. Please wait and try again."
    except openai.error.OpenAIError as e:
        return f"An OpenAI API error occurred: {e}"
    except Exception as e:
        return f"An unexpected error occurred: {e}"

def generate_text_completion(prompt_text, engine="text-davinci-003",
temperature=0.7, max_tokens=150):
    """
    Generates text using OpenAI's older Completion API.
    Note: 'text-davinci-003' is being deprecated. Use chat models where
possible.
    """
    try:
        response = openai.Completion.create(
            engine=engine,
            prompt=prompt_text,
            temperature=temperature,
            max_tokens=max_tokens
        )
        return response.choices[0].text.strip()
    except openai.error.AuthenticationError:
        return "Error: Invalid OpenAI API key. Please check your key."
    except openai.error.RateLimitError:
        return "Error: Rate limit exceeded. Please wait and try again."
    except openai.error.OpenAIError as e:
        return f"An OpenAI API error occurred: {e}"
    except Exception as e:
        return f"An unexpected error occurred: {e}"

# --- Main Execution ---
if __name__ == "__main__":
    print("--- Text Generation using Chat Completion API (gpt-3.5-turbo) ---
")
    prompt1 = "Write a short poem about a rainy day."
    generated_poem = generate_text_chat_completion(prompt1, model="gpt-3.5-
turbo")
    print(f"Prompt: '{prompt1}'")
    print(f"Generated Text:\n{generated_poem}\n")

    print("--- Text Generation using Chat Completion API (gpt-4o) ---")
    prompt2 = "Summarize the plot of 'Alice in Wonderland' in two sentences."
```

```
    generated_summary = generate_text_chat_completion(prompt2, model="gpt-
4o", temperature=0.3)
    print(f"Prompt: '{prompt2}'")
    print(f"Generated Text:\n{generated_summary}\n")

    print("--- Text Generation using Completion API (text-davinci-003 - if
available) ---")
    print("Note: 'text-davinci-003' is an older model and may not be
available or recommended.")
    prompt3 = "List five benefits of learning Python programming."
    generated_list = generate_text_completion(prompt3, engine="text-davinci-
003", max_tokens=100)
    print(f"Prompt: '{prompt3}'")
    print(f"Generated Text:\n{generated_list}\n")
```

Input

The input for this program is the `prompt_text` string passed to the `generate_text_chat_completion` or `generate_text_completion` functions. You can modify these prompts to generate different types of text.

Example Input:

- `"Write a short poem about a rainy day."`
- `"Summarize the plot of 'Alice in Wonderland' in two sentences."`
- `"List five benefits of learning Python programming."`

Expected Output

The program will print the original prompt and the text generated by the OpenAI model in response to that prompt. The exact generated text will vary each time due to the probabilistic nature of LLMs and the `temperature` setting.

Example Expected Output:

```
--- Text Generation using Chat Completion API (gpt-3.5-turbo) ---
Prompt: 'Write a short poem about a rainy day.'
Generated Text:
Soft drops fall, a gentle beat,
Washing clean the dusty street.
Grey skies weep, a whispered sigh,
Nature's tears from up on high.
Cozy warmth, a quiet grace,
Rainy day, a peaceful space.

--- Text Generation using Chat Completion API (gpt-4o) ---
Prompt: 'Summarize the plot of 'Alice in Wonderland' in two sentences."
Generated Text:
Alice tumbles down a rabbit hole into a fantastical world filled with
peculiar creatures and illogical events. She navigates through bizarre
encounters, including a mad tea party and a croquet game with the Queen of
Hearts, all while trying to find her way home.

--- Text Generation using Completion API (text-davinci-003 - if available) --
-
Note: 'text-davinci-003' is an older model and may not be available or
recommended.
Prompt: 'List five benefits of learning Python programming.'
Generated Text:
```

1. Versatility: Python can be used for web development, data analysis, AI, machine learning, and more.
2. Easy to Learn: Its simple syntax and readability make it beginner-friendly.
3. Large Community and Libraries: Extensive support and a vast ecosystem of libraries (e.g., NumPy, Pandas, TensorFlow) accelerate development.
4. High Demand: Python skills are highly sought after in the job market across various industries.
5. Automation: Excellent for scripting and automating repetitive tasks, saving time and effort.

Lab 9: How to generate text using Lang Chain and OpenAI

Title

Text Generation using LangChain and OpenAI

Aim

To demonstrate how to use the LangChain framework to interact with OpenAI's Large Language Models for text generation, showcasing basic prompt templating and chaining.

Procedure

1. **Setup Environment:**
    - Install necessary libraries: `langchain` and `openai`.
    - Set your OpenAI API key (as an environment variable is recommended).
2. **Initialize LLM:**
    - Import the `ChatOpenAI` class from `langchain.chat_models`.
    - Initialize an instance of `ChatOpenAI` with your desired model and parameters (e.g., `temperature`).
3. **Create Prompt Template:**
    - Use `PromptTemplate` from `langchain.prompts` to define a reusable template for your prompts, allowing for dynamic insertion of variables.
4. **Create Chains (Optional but Recommended):**
    - Use `LLMChain` from `langchain.chains` to combine an LLM with a prompt template, creating a single callable object for text generation.
    - (Advanced) Explore `SimpleSequentialChain` or other chain types for more complex workflows.
5. **Invoke Chain:**
    - Call the `run` method on the created chain, passing the necessary input variables.
6. **Process Output:**
    - Retrieve and display the generated text.
    - Handle potential errors.

Source Code

```
import os
from langchain.chat_models import ChatOpenAI
from langchain.prompts import PromptTemplate
from langchain.chains import LLMChain, SimpleSequentialChain

# --- Configuration ---
# Set your OpenAI API key as an environment variable
# e.g., export OPENAI_API_KEY="YOUR_API_KEY_HERE"
# If not set, replace "YOUR_OPENAI_API_KEY" with your actual key for testing
purposes.
OPENAI_API_KEY = os.getenv("OPENAI_API_KEY", "YOUR_OPENAI_API_KEY")

if OPENAI_API_KEY == "YOUR_OPENAI_API_KEY":
    print("WARNING: OpenAI API Key not set as environment variable
'OPENAI_API_KEY'.")
    print("Please set it or replace 'YOUR_OPENAI_API_KEY' in the script.")
    # In a real application, you might exit or raise an error here.

# --- Initialize LLM ---
# Using gpt-3.5-turbo as a common choice. Adjust model and temperature as
needed.
```

```
try:
    llm = ChatOpenAI(openai_api_key=OPENAI_API_KEY, model_name="gpt-3.5-
turbo", temperature=0.7)
    print(f"Initialized LLM: {llm.model_name}")
except Exception as e:
    print(f"Error initializing LLM: {e}")
    llm = None # Set llm to None if initialization fails

def generate_text_with_langchain(topic, style):
    """
    Generates text using LangChain with a single LLMChain.
    """
    if not llm:
        return "LLM not initialized due to API key error. Cannot generate
text."

    # 1. Create Prompt Template
    # Define how the user's input will be structured for the LLM
    prompt_template = PromptTemplate(
        input_variables=["topic", "style"],
        template="Write a short paragraph about {topic} in a {style} style."
    )

    # 2. Create LLMChain
    # Combine the LLM and the prompt template
    text_generation_chain = LLMChain(llm=llm, prompt=prompt_template)

    # 3. Invoke Chain
    try:
        response = text_generation_chain.run(topic=topic, style=style)
        return response
    except Exception as e:
        return f"An error occurred during text generation: {e}"

def generate_text_with_sequential_chain(product_name):
    """
    Demonstrates a simple sequential chain:
    1. Generate a product description.
    2. Generate a catchy slogan based on the description.
    """
    if not llm:
        return "LLM not initialized due to API key error. Cannot generate
text."

    # Chain 1: Generate product description
    description_prompt = PromptTemplate(
        input_variables=["product_name"],
        template="Write a concise and appealing product description for a new
{product_name}."
    )
    description_chain = LLMChain(llm=llm, prompt=description_prompt,
output_key="product_description")

    # Chain 2: Generate slogan based on description
    slogan_prompt = PromptTemplate(
        input_variables=["product_description"],
        template="Based on the following product description, create a catchy
advertising slogan:\n\n{product_description}"
    )
    slogan_chain = LLMChain(llm=llm, prompt=slogan_prompt,
output_key="slogan")

    # Combine into a SimpleSequentialChain
    overall_chain = SimpleSequentialChain(chains=[description_chain,
slogan_chain], verbose=True)
```

```
    try:
        response = overall_chain.run(product_name)
        return response # The output of the last chain in the sequence
    except Exception as e:
        return f"An error occurred during sequential text generation: {e}"

# --- Main Execution ---
if __name__ == "__main__":
    print("--- Single Chain Text Generation ---")
    topic1 = "the benefits of meditation"
    style1 = "inspirational"
    generated_text1 = generate_text_with_langchain(topic1, style1)
    print(f"Prompt: Write a paragraph about '{topic1}' in an '{style1}'
style.")
    print(f"Generated Text:\n{generated_text1}\n")

    topic2 = "the history of artificial intelligence"
    style2 = "concise academic"
    generated_text2 = generate_text_with_langchain(topic2, style2)
    print(f"Prompt: Write a paragraph about '{topic2}' in a '{style2}'
style.")
    print(f"Generated Text:\n{generated_text2}\n")

    print("\n--- Sequential Chain Text Generation ---")
    product = "smartwatch with health monitoring"
    generated_slogan = generate_text_with_sequential_chain(product)
    print(f"Product: '{product}'")
    print(f"Generated Slogan:\n{generated_slogan}\n")
```

Input

The input for this program is defined by the variables passed to the LangChain functions. For `generate_text_with_langchain`, it's `topic` and `style`. For `generate_text_with_sequential_chain`, it's `product_name`.

Example Input:

- `topic="the benefits of meditation", style="inspirational"`
- `topic="the history of artificial intelligence", style="concise academic"`
- `product_name="smartwatch with health monitoring"`

Expected Output

The program will print the generated text based on the specified topic and style for the single chain example. For the sequential chain, it will first show the intermediate product description generated and then the final catchy slogan.

Example Expected Output:

```
Initialized LLM: gpt-3.5-turbo

--- Single Chain Text Generation ---
Prompt: Write a paragraph about 'the benefits of meditation' in an
'inspirational' style.
Generated Text:
Embrace the profound tranquility that meditation offers, a sacred pause in
life's relentless rhythm. It's an invitation to quiet the mind's incessant
chatter, allowing inner wisdom to surface. Through this practice, you
cultivate a serene sanctuary within, fostering clarity, reducing stress, and
```

nurturing a deeper connection to your true self. Let each breath be a gentle anchor, guiding you towards a more peaceful, resilient, and joyful existence.

Prompt: Write a paragraph about 'the history of artificial intelligence' in a 'concise academic' style.
Generated Text:
The genesis of artificial intelligence traces back to ancient myths of intelligent automata, but its modern foundations were laid in the mid-20th century with the advent of electronic computers and the Dartmouth workshop in 1956. Early AI research focused on symbolic reasoning, expert systems, and problem-solving, experiencing periods of both optimism and "AI winters." The 21st century has witnessed a resurgence driven by advancements in computational power, vast datasets, and novel algorithms like deep learning, leading to significant breakthroughs in areas such as natural language processing and computer vision.

--- Sequential Chain Text Generation ---
> Entering new SimpleSequentialChain chain...
Product: 'smartwatch with health monitoring'
> Finished chain.
Generated Slogan:
"Your Health, Your Time, Your Smartwatch."

Lab 10: Text Summarization using LLM

Title

Text Summarization using Large Language Models (LLMs)

Aim

To implement a Python program that utilizes a Large Language Model (LLM) (e.g., from OpenAI) to perform abstractive text summarization on a given input text.

Procedure

1. **Setup LLM Access:**
    - Ensure the `openai` library is installed and your API key is configured.
2. **Define Summarization Prompt:**
    - Craft an effective prompt that instructs the LLM to summarize the provided text. The prompt should specify the desired length, style, and focus of the summary.
    - Example: `"Summarize the following text concisely in 3 sentences:"`
3. **Input Text Preparation:**
    - Provide the long text you wish to summarize.
4. **LLM Interaction:**
    - Combine the summarization instruction with the input text into a single prompt for the LLM.
    - Call the LLM API (e.g., `openai.ChatCompletion.create`) with the prepared prompt.
    - Specify parameters like `max_tokens` to control the summary length.
5. **Extract and Display Summary:**
    - Parse the LLM's response to extract the generated summary.
    - Print the original text and its summary.
    - Implement error handling for API issues.

Source Code

```
import openai
import os

# --- Configuration ---
# Set your OpenAI API key as an environment variable
# e.g., export OPENAI_API_KEY="YOUR_API_KEY_HERE"
# If not set, replace "YOUR_OPENAI_API_KEY" with your actual key for testing
purposes.
OPENAI_API_KEY = os.getenv("OPENAI_API_KEY", "YOUR_OPENAI_API_KEY")

if OPENAI_API_KEY == "YOUR_OPENAI_API_KEY":
    print("WARNING: OpenAI API Key not set as environment variable
'OPENAI_API_KEY'.")
    print("Please set it or replace 'YOUR_OPENAI_API_KEY' in the script.")
    # In a real application, you might exit or raise an error here.

openai.api_key = OPENAI_API_KEY

def summarize_text_with_llm(text_to_summarize, model="gpt-3.5-turbo",
max_tokens=100, summary_length="3 sentences"):
    """
    Summarizes the given text using an OpenAI LLM.
    """
    if not openai.api_key:
```

```
        return "Error: OpenAI API Key is not configured."

    prompt = f"Summarize the following text concisely in
{summary_length}:\n\nTEXT:\n{text_to_summarize}\n\nSUMMARY:"

    try:
        response = openai.ChatCompletion.create(
            model=model,
            messages=[
                {"role": "system", "content": "You are a helpful assistant
specialized in text summarization."},
                {"role": "user", "content": prompt}
            ],
            temperature=0.3, # Keep temperature low for factual summaries
            max_tokens=max_tokens
        )
        return response.choices[0].message['content'].strip()
    except openai.error.AuthenticationError:
        return "Error: Invalid OpenAI API key. Please check your key."
    except openai.error.RateLimitError:
        return "Error: Rate limit exceeded. Please wait and try again."
    except openai.error.OpenAIError as e:
        return f"An OpenAI API error occurred: {e}"
    except Exception as e:
        return f"An unexpected error occurred: {e}"

# --- Main Execution ---
if __name__ == "__main__":
    long_text_1 = """
    Artificial intelligence (AI) is a rapidly expanding field that aims to
create machines capable of performing tasks that typically require human
intelligence. These tasks include learning, problem-solving, decision-making,
perception, and understanding language. AI encompasses various subfields,
such as machine learning, deep learning, natural language processing (NLP),
and computer vision. Machine learning, a core component of modern AI,
involves training algorithms on data to make predictions or decisions without
being explicitly programmed. Deep learning, a subset of machine learning,
uses neural networks with many layers to learn complex patterns from large
datasets. The applications of AI are vast and growing, ranging from self-
driving cars and medical diagnosis to financial trading and personalized
recommendations. While AI offers immense potential for societal advancement,
it also raises important ethical considerations regarding privacy, bias, and
job displacement.
    """

    long_text_2 = """
    The Amazon rainforest is the largest rainforest in the world, covering an
immense area across nine South American countries, primarily Brazil. It is
renowned for its unparalleled biodiversity, housing an estimated 10% of the
world's known species, including countless insects, plants, birds, and
mammals. The Amazon plays a crucial role in regulating the Earth's climate by
absorbing vast amounts of carbon dioxide and releasing oxygen, often referred
to as the "lungs of the Earth." However, this vital ecosystem faces severe
threats from deforestation, primarily due to agricultural expansion, logging,
and mining. The loss of the Amazon rainforest has significant implications
for global climate change, indigenous communities, and the survival of
numerous species. Conservation efforts are underway, but the challenge
remains immense.
    """

    print("--- Summarization Example 1 ---")
    print("Original Text:")
    print(long_text_1)
    summary_1 = summarize_text_with_llm(long_text_1, summary_length="2
sentences", max_tokens=60)
    print("\nSummary (2 sentences):")
```

```
    print(summary_1)

    print("\n" + "="*50 + "\n")

    print("--- Summarization Example 2 ---")
    print("Original Text:")
    print(long_text_2)
    summary_2 = summarize_text_with_llm(long_text_2, summary_length="a short
paragraph", max_tokens=120)
    print("\nSummary (a short paragraph):")
    print(summary_2)
```

Input

The input for this program is the `text_to_summarize` string, which is the longer document you want to condense. You can also specify the `summary_length` (e.g., "2 sentences", "a short paragraph") and `max_tokens` to control the output.

Example Input:

- A long paragraph about Artificial Intelligence.
- A detailed description of the Amazon rainforest.

Expected Output

The program will print the original long text and then the summarized version generated by the LLM, adhering to the specified length and style.

Example Expected Output:

```
--- Summarization Example 1 ---
Original Text:
    Artificial intelligence (AI) is a rapidly expanding field that aims to
create machines capable of performing tasks that typically require human
intelligence. These tasks include learning, problem-solving, decision-making,
perception, and understanding language. AI encompasses various subfields,
such as machine learning, deep learning, natural language processing (NLP),
and computer vision. Machine learning, a core component of modern AI,
involves training algorithms on data to make predictions or decisions without
being explicitly programmed. Deep learning, a subset of machine learning,
uses neural networks with many layers to learn complex patterns from large
datasets. The applications of AI are vast and growing, ranging from self-
driving cars and medical diagnosis to financial trading and personalized
recommendations. While AI offers immense potential for societal advancement,
it also raises important ethical considerations regarding privacy, bias, and
job displacement.

Summary (2 sentences):
Artificial intelligence (AI) is a growing field focused on creating machines
with human-like intelligence for tasks like learning and problem-solving,
encompassing subfields such as machine learning and deep learning. Its vast
applications, from self-driving cars to medical diagnosis, come with ethical
considerations regarding privacy, bias, and job displacement.

==================================================

--- Summarization Example 2 ---
Original Text:
    The Amazon rainforest is the largest rainforest in the world, covering an
immense area across nine South American countries, primarily Brazil. It is
renowned for its unparalleled biodiversity, housing an estimated 10% of the
```

world's known species, including countless insects, plants, birds, and mammals. The Amazon plays a crucial role in regulating the Earth's climate by absorbing vast amounts of carbon dioxide and releasing oxygen, often referred to as the "lungs of the Earth." However, this vital ecosystem faces severe threats from deforestation, primarily due to agricultural expansion, logging, and mining. The loss of the Amazon rainforest has significant implications for global climate change, indigenous communities, and the survival of numerous species. Conservation efforts are underway, but the challenge remains immense.

Summary (a short paragraph):
The Amazon rainforest, the world's largest, spans nine South American countries and boasts unparalleled biodiversity, hosting an estimated 10% of global species. Crucial for climate regulation by absorbing carbon dioxide, it is often called the "lungs of the Earth." However, it faces severe deforestation threats from agriculture, logging, and mining, with significant implications for global climate, indigenous populations, and species survival, making ongoing conservation efforts vital.

Lab 11: Sentimental analysis using LLM

Title

Sentiment Analysis using Large Language Models (LLMs)

Aim

To implement a Python program that leverages a Large Language Model (LLM) (e.g., from OpenAI) to determine the sentiment (e.g., positive, negative, neutral) of a given piece of text.

Procedure

1. **Setup LLM Access:**
    - Ensure the `openai` library is installed and your API key is configured.
2. **Define Sentiment Analysis Prompt:**
    - Craft a clear prompt that instructs the LLM to analyze the sentiment of the provided text. Specify the desired output format (e.g., "Positive", "Negative", "Neutral").
    - Example: `"Analyze the sentiment of the following text and classify it as 'Positive', 'Negative', or 'Neutral':"`
3. **Input Text Preparation:**
    - Provide the text for which you want to determine the sentiment.
4. **LLM Interaction:**
    - Combine the sentiment analysis instruction with the input text into a single prompt for the LLM.
    - Call the LLM API (e.g., `openai.ChatCompletion.create`).
    - Consider using a low `temperature` to encourage deterministic and factual responses.
5. **Extract and Display Sentiment:**
    - Parse the LLM's response to extract the predicted sentiment.
    - Print the original text and its determined sentiment.
    - Implement error handling for API issues.

Source Code

```
import openai
import os

# --- Configuration ---
# Set your OpenAI API key as an environment variable
# e.g., export OPENAI_API_KEY="YOUR_API_KEY_HERE"
# If not set, replace "YOUR_OPENAI_API_KEY" with your actual key for testing
purposes.
OPENAI_API_KEY = os.getenv("OPENAI_API_KEY", "YOUR_OPENAI_API_KEY")

if OPENAI_API_KEY == "YOUR_OPENAI_API_KEY":
    print("WARNING: OpenAI API Key not set as environment variable
'OPENAI_API_KEY'.")
    print("Please set it or replace 'YOUR_OPENAI_API_KEY' in the script.")
    # In a real application, you might exit or raise an error here.

openai.api_key = OPENAI_API_KEY

def analyze_sentiment_with_llm(text_to_analyze, model="gpt-3.5-turbo"):
    """
    Analyzes the sentiment of the given text using an OpenAI LLM.
    Returns 'Positive', 'Negative', 'Neutral', or an error message.
```

```
    """
    if not openai.api_key:
        return "Error: OpenAI API Key is not configured."

    prompt = f"""
    Analyze the sentiment of the following text.
    Classify the sentiment as 'Positive', 'Negative', or 'Neutral'.
    Provide only the sentiment label as the output.

    TEXT:
    {text_to_analyze}
    """

    try:
        response = openai.ChatCompletion.create(
            model=model,
            messages=[
                {"role": "system", "content": "You are a sentiment analysis
expert. Respond only with 'Positive', 'Negative', or 'Neutral'."},
                {"role": "user", "content": prompt}
            ],
            temperature=0.0, # Keep temperature very low for deterministic
classification
            max_tokens=10 # Expecting a short output (e.g., "Positive")
        )
        sentiment = response.choices[0].message['content'].strip()
        # Basic validation to ensure the output is one of the expected labels
        if sentiment in ['Positive', 'Negative', 'Neutral']:
            return sentiment
        else:
            return f"LLM returned unexpected output: '{sentiment}'"
    except openai.error.AuthenticationError:
        return "Error: Invalid OpenAI API key. Please check your key."
    except openai.error.RateLimitError:
        return "Error: Rate limit exceeded. Please wait and try again."
    except openai.error.OpenAIError as e:
        return f"An OpenAI API error occurred: {e}"
    except Exception as e:
        return f"An unexpected error occurred: {e}"

# --- Main Execution ---
if __name__ == "__main__":
    texts_for_analysis = [
        "I absolutely loved the new movie! The acting was superb and the plot
was engaging.",
        "The customer service was terrible, and the product broke after just
one day.",
        "The weather today is neither good nor bad, just cloudy.",
        "This is an incredibly insightful article, well-researched and
clearly written.",
        "The delivery was late, and the food was cold. Very disappointing.",
        "The book was okay, nothing special, but not bad either."
    ]

    for text in texts_for_analysis:
        print(f"Text: '{text}'")
        sentiment = analyze_sentiment_with_llm(text)
        print(f"Sentiment: {sentiment}\n")
```

Input

The input for this program is the `text_to_analyze` string, which is the piece of text for which you want to determine the sentiment.

Example Input:

- `"I absolutely loved the new movie! The acting was superb and the plot was engaging."`
- `"The customer service was terrible, and the product broke after just one day."`
- `"The weather today is neither good nor bad, just cloudy."`
- `"This is an incredibly insightful article, well-researched and clearly written."`

Expected Output

The program will print each input text followed by its determined sentiment (Positive, Negative, or Neutral) as classified by the LLM.

Example Expected Output:

```
Text: 'I absolutely loved the new movie! The acting was superb and the plot
was engaging.'
Sentiment: Positive

Text: 'The customer service was terrible, and the product broke after just
one day.'
Sentiment: Negative

Text: 'The weather today is neither good nor bad, just cloudy.'
Sentiment: Neutral

Text: 'This is an incredibly insightful article, well-researched and clearly
written.'
Sentiment: Positive

Text: 'The delivery was late, and the food was cold. Very disappointing.'
Sentiment: Negative

Text: 'The book was okay, nothing special, but not bad either.'
Sentiment: Neutral
```

Lab 12: Using Pre-trained BERT model for Summarization

Title

Text Summarization using Pre-trained BERT Model

Aim

To implement abstractive text summarization using a pre-trained BERT-based model (e.g., `bert-large-uncased-whole-word-masking-finetuned-squad` or a dedicated summarization model like `t5-base`) from the Hugging Face Transformers library.

Procedure

1. **Setup Environment:**
    - Install the `transformers` and `torch` (or `tensorflow`) libraries.
2. **Load Pre-trained Model and Tokenizer:**
    - Import `AutoTokenizer` and `AutoModelForSeq2SeqLM` (for sequence-to-sequence models like T5/BART, which are good for abstractive summarization) or `pipeline` for a simpler approach.
    - Load a pre-trained summarization model (e.g., `t5-base`) and its corresponding tokenizer.
3. **Prepare Input Text:**
    - Provide the long text you want to summarize.
    - Tokenize the input text using the loaded tokenizer.
    - Ensure the input is within the model's maximum sequence length (truncate if necessary).
4. **Generate Summary:**
    - Pass the tokenized input to the model's `generate` method.
    - Specify generation parameters like `min_length`, `max_length`, `num_beams` (for beam search decoding), and `do_sample=False` (for deterministic output).
5. **Decode and Display:**
    - Decode the generated token IDs back into human-readable text using the tokenizer.
    - Print the original text and its summary.

Source Code

```
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM, pipeline
import torch

# --- Configuration ---
# Choose a model suitable for summarization. T5 is a good choice for
abstractive summarization.
# Other options: 'sshleifer/distilbart-cnn-12-6', 'facebook/bart-large-cnn'
MODEL_NAME = "t5-base"

# --- 1. Setup Environment & Load Model/Tokenizer ---
print(f"Loading tokenizer and model for {MODEL_NAME}...")
try:
    # For sequence-to-sequence models like T5/BART, AutoModelForSeq2SeqLM is
appropriate.
    tokenizer = AutoTokenizer.from_pretrained(MODEL_NAME)
    model = AutoModelForSeq2SeqLM.from_pretrained(MODEL_NAME)

    # Alternatively, use the pipeline for simpler usage:
    # summarizer_pipeline = pipeline("summarization", model=MODEL_NAME)
```

```
    print("Model and tokenizer loaded successfully.")
except Exception as e:
    print(f"Error loading model or tokenizer: {e}")
    tokenizer = None
    model = None
    # summarizer_pipeline = None # If using pipeline

def summarize_text_with_bert_like_model(text_to_summarize, min_len=30,
max_len=150):
    """
    Summarizes the given text using a pre-trained T5 (BERT-like) model.
    """
    if tokenizer is None or model is None:
        return "Error: Model or tokenizer not loaded. Cannot summarize."

    # For T5, input typically needs a prefix like "summarize: "
    input_text = "summarize: " + text_to_summarize

    # 2. Prepare Input Text
    # Tokenize and encode the input text
    inputs = tokenizer(
        input_text,
        return_tensors="pt",
        max_length=512, # T5's default max input length is 512
        truncation=True
    )

    # 3. Generate Summary
    # Generate the summary using the model
    summary_ids = model.generate(
        inputs["input_ids"],
        num_beams=4, # Use beam search for better quality summaries
        min_length=min_len,
        max_length=max_len,
        early_stopping=True # Stop when all beam hypotheses have reached EOS
token
    )

    # 4. Decode and Display
    # Decode the generated token IDs back to text
    summary = tokenizer.decode(summary_ids[0], skip_special_tokens=True)
    return summary

# --- Main Execution ---
if __name__ == "__main__":
    long_text_1 = """
    Artificial intelligence (AI) is a rapidly expanding field that aims to
create machines capable of performing tasks that typically require human
intelligence. These tasks include learning, problem-solving, decision-making,
perception, and understanding language. AI encompasses various subfields,
such as machine learning, deep learning, natural language processing (NLP),
and computer vision. Machine learning, a core component of modern AI,
involves training algorithms on data to make predictions or decisions without
being explicitly programmed. Deep learning, a subset of machine learning,
uses neural networks with many layers to learn complex patterns from large
datasets. The applications of AI are vast and growing, ranging from self-
driving cars and medical diagnosis to financial trading and personalized
recommendations. While AI offers immense potential for societal advancement,
it also raises important ethical considerations regarding privacy, bias, and
job displacement.
    """

    long_text_2 = """
    The Amazon rainforest is the largest rainforest in the world, covering an
immense area across nine South American countries, primarily Brazil. It is
renowned for its unparalleled biodiversity, housing an estimated 10% of the
```

```
world's known species, including countless insects, plants, birds, and
mammals. The Amazon plays a crucial role in regulating the Earth's climate by
absorbing vast amounts of carbon dioxide and releasing oxygen, often referred
to as the "lungs of the Earth." However, this vital ecosystem faces severe
threats from deforestation, primarily due to agricultural expansion, logging,
and mining. The loss of the Amazon rainforest has significant implications
for global climate change, indigenous communities, and the survival of
numerous species. Conservation efforts are underway, but the challenge
remains immense.
    """

    print("--- Summarization Example 1 (T5-base) ---")
    print("Original Text:")
    print(long_text_1)
    summary_1 = summarize_text_with_bert_like_model(long_text_1, min_len=40,
max_len=80)
    print("\nSummary:")
    print(summary_1)

    print("\n" + "="*50 + "\n")

    print("--- Summarization Example 2 (T5-base) ---")
    print("Original Text:")
    print(long_text_2)
    summary_2 = summarize_text_with_bert_like_model(long_text_2, min_len=50,
max_len=100)
    print("\nSummary:")
    print(summary_2)

    # --- Example using pipeline (simpler, but less control) ---
    # if summarizer_pipeline:
    #     print("\n--- Summarization Example 3 (using pipeline) ---")
    #     print("Original Text:")
    #     print(long_text_1)
    #     pipeline_summary = summarizer_pipeline(long_text_1, min_length=40,
max_length=80, do_sample=False)
    #     print("\nSummary (from pipeline):")
    #     print(pipeline_summary[0]['summary_text'])
```

Input

The input for this program is the `text_to_summarize` string, which is the longer document you want to condense. You can also adjust `min_len` and `max_len` to control the length of the generated summary.

Example Input:

- A long paragraph about Artificial Intelligence.
- A detailed description of the Amazon rainforest.

Expected Output

The program will print the original long text and then its abstractive summary generated by the pre-trained BERT-based model (T5 in this case). The summary will be a condensed version of the original text, potentially using new phrasing.

Example Expected Output:

```
Loading tokenizer and model for t5-base...
Model and tokenizer loaded successfully.
```

--- Summarization Example 1 (T5-base) ---
Original Text:
    Artificial intelligence (AI) is a rapidly expanding field that aims to create machines capable of performing tasks that typically require human intelligence. These tasks include learning, problem-solving, decision-making, perception, and understanding language. AI encompasses various subfields, such as machine learning, deep learning, natural language processing (NLP), and computer vision. Machine learning, a core component of modern AI, involves training algorithms on data to make predictions or decisions without being explicitly programmed. Deep learning, a subset of machine learning, uses neural networks with many layers to learn complex patterns from large datasets. The applications of AI are vast and growing, ranging from self-driving cars and medical diagnosis to financial trading and personalized recommendations. While AI offers immense potential for societal advancement, it also raises important ethical considerations regarding privacy, bias, and job displacement.

Summary:
AI is a rapidly expanding field that aims to create machines capable of performing tasks that typically require human intelligence. It encompasses various subfields, such as machine learning, deep learning, natural language processing (NLP), and computer vision. The applications of AI are vast and growing, ranging from self-driving cars and medical diagnosis to financial trading and personalized recommendations.

==

--- Summarization Example 2 (T5-base) ---
Original Text:
    The Amazon rainforest is the largest rainforest in the world, covering an immense area across nine South American countries, primarily Brazil. It is renowned for its unparalleled biodiversity, housing an estimated 10% of the world's known species, including countless insects, plants, birds, and mammals. The Amazon plays a crucial role in regulating the Earth's climate by absorbing vast amounts of carbon dioxide and releasing oxygen, often referred to as the "lungs of the Earth." However, this vital ecosystem faces severe threats from deforestation, primarily due to agricultural expansion, logging, and mining. The loss of the Amazon rainforest has significant implications for global climate change, indigenous communities, and the survival of numerous species. Conservation efforts are underway, but the challenge remains immense.

Summary:
The Amazon rainforest is the largest rainforest in the world, covering an immense area across nine South American countries. It is renowned for its unparalleled biodiversity, housing an estimated 10% of the world's known species. The Amazon plays a crucial role in regulating the Earth's climate by absorbing vast amounts of carbon dioxide and releasing oxygen.

Lab 13: Chatbot and Virtual Assistance

Title

Building a Simple Chatbot and Virtual Assistant

Aim

To develop a basic chatbot and virtual assistant using Python, capable of understanding simple user queries and providing relevant responses. This lab can be extended using rule-based systems, pattern matching, or integrating with LLMs for more advanced capabilities.

Procedure

1. **Define Intentions and Responses (Rule-based):**
    - Create a dictionary or a list of patterns and corresponding responses to handle common user queries (e.g., greetings, questions about time, simple facts).
2. **Input Processing:**
    - Accept user input.
    - Preprocess the input (e.g., convert to lowercase, remove punctuation) to make pattern matching easier.
3. **Pattern Matching/Intent Recognition:**
    - Implement logic to match the preprocessed user input against predefined patterns or keywords to identify the user's intent.
    - (Optional for advanced) Use libraries like `nltk` for more sophisticated tokenization or `spaCy` for named entity recognition.
4. **Response Generation:**
    - Based on the recognized intent, select and return an appropriate response.
    - Handle cases where the intent is not recognized (fallback responses).
5. **Conversation Loop:**
    - Create a loop that continuously prompts the user for input, processes it, and provides a response until the user decides to exit.

Source Code

```
import re
import datetime

class SimpleChatbot:
    def __init__(self):
        self.responses = {
            "greeting": ["Hello there!", "Hi! How can I help you?",
"Greetings!"],
            "farewell": ["Goodbye!", "See you later!", "Farewell!"],
            "thanks": ["You're welcome!", "No problem!", "Glad to help!"],
            "name_query": ["I am a simple chatbot, you can call me Bot.", "I
don't have a name, but I'm here to assist you."],
            "time_query": self._get_current_time, # Function reference
            "date_query": self._get_current_date, # Function reference
            "weather_query": ["I cannot check the weather right now.", "I
don't have access to real-time weather data."],
            "default": ["I'm not sure I understand.", "Could you please
rephrase that?", "Can you tell me more?"]
        }
        self.patterns = {
            r".*hello|hi|hey.*": "greeting",
            r".*bye|goodbye|see you.*": "farewell",
            r".*thank you|thanks.*": "thanks",
```

```
            r".*your name|who are you.*": "name_query",
            r".*time is it|current time.*": "time_query",
            r".*date today|current date.*": "date_query",
            r".*weather like|how's the weather.*": "weather_query",
        }

    def _get_current_time(self):
        """Returns the current time."""
        now = datetime.datetime.now()
        return f"The current time is {now.strftime('%H:%M:%S')}."

    def _get_current_date(self):
        """Returns the current date."""
        today = datetime.date.today()
        return f"Today's date is {today.strftime('%Y-%m-%d')}."

    def get_response(self, user_input):
        """
        Matches user input to a predefined pattern and returns a response.
        """
        user_input = user_input.lower()

        for pattern, intent in self.patterns.items():
            if re.search(pattern, user_input):
                response_data = self.responses.get(intent)
                if callable(response_data): # If it's a function, call it
                    return response_data()
                else: # Otherwise, pick a random response from the list
                    return np.random.choice(response_data)

        # If no pattern matches, return a default response
        return np.random.choice(self.responses["default"])

    def start_chat(self):
        """Starts the interactive chatbot session."""
        print("Chatbot: Hello! Type 'exit' to end the conversation.")
        while True:
            user_input = input("You: ")
            if user_input.lower() == 'exit':
                print("Chatbot: Goodbye!")
                break
            response = self.get_response(user_input)
            print(f"Chatbot: {response}")

# --- Main Execution ---
if __name__ == "__main__":
    import numpy as np # Import here for random.choice

    chatbot = SimpleChatbot()
    chatbot.start_chat()
```

Input

The input for this program is interactive, provided by the user through the console. Users type their queries, and the chatbot responds.

Example Input (user types):

- Hello
- What is your name?
- What time is it?
- How's the weather?
- Thank you

- `Goodbye`
- `Tell me a joke` (This would trigger a default response)
- `exit` (to end the chat)

Expected Output

The program will simulate a conversation with the chatbot. It will respond to recognized patterns with predefined or dynamically generated answers (like current time/date) and provide default responses for unrecognized queries.

Example Expected Output:

```
Chatbot: Hello! Type 'exit' to end the conversation.
You: Hi there!
Chatbot: Hello there!
You: What is your name?
Chatbot: I am a simple chatbot, you can call me Bot.
You: What time is it?
Chatbot: The current time is HH:MM:SS. (e.g., 14:35:22)
You: How's the weather outside?
Chatbot: I cannot check the weather right now.
You: Thank you so much!
Chatbot: Glad to help!
You: Tell me something interesting.
Chatbot: I'm not sure I understand.
You: exit
Chatbot: Goodbye!
```

Lab 14: Movie Prediction

Title

Movie Prediction (Recommendation System)

Aim

To implement a basic movie prediction or recommendation system using collaborative filtering (user-based or item-based) or a content-based approach. For this lab, we'll focus on a simple content-based approach using movie genres.

Procedure

1. **Dataset Preparation:**
    - Create a small dataset of movies with features like `title`, `genre`, and `description`.
    - Represent genres as numerical vectors (e.g., one-hot encoding or multi-hot encoding).
2. **User Profile/Preference:**
    - Define a user's preferences, perhaps by listing movies they liked or genres they prefer.
3. **Similarity Calculation:**
    - For content-based filtering, calculate the similarity between a user's preferred genres and the genres of unrated movies. Cosine similarity is a common metric.
4. **Recommendation Generation:**
    - Sort unrated movies by their similarity scores to the user's preferences.
    - Recommend the top-N most similar movies.
5. **Implementation:**
    - Use libraries like `pandas` for data handling and `sklearn.metrics.pairwise` for similarity calculations.

Source Code

```
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity

class MovieRecommender:
    def __init__(self, movies_data):
        self.movies_df = pd.DataFrame(movies_data)
        self.movies_df['genres_list'] = self.movies_df['genre'].apply(lambda
x: [g.strip().lower() for g in x.split(',')])

        # Create a TF-IDF vectorizer for genres
        # This will convert genre strings into numerical vectors
        self.tfidf_vectorizer = TfidfVectorizer(tokenizer=lambda x: x,
lowercase=False)
        self.genre_matrix =
self.tfidf_vectorizer.fit_transform(self.movies_df['genres_list'])

        print("Movie data loaded and genre matrix created.")
        print(f"Available movies: {self.movies_df['title'].tolist()}")

    def get_movie_genre_vector(self, movie_title):
        """Returns the genre vector for a given movie title."""
        if movie_title not in self.movies_df['title'].values:
            print(f"Error: Movie '{movie_title}' not found.")
```

```
            return None

        idx = self.movies_df[self.movies_df['title'] == movie_title].index[0]
        return self.genre_matrix[idx]

    def recommend_movies(self, liked_movies, num_recommendations=5):
        """
        Recommends movies based on a list of liked movies using content-based
filtering (genres).
        """
        if not liked_movies:
            print("Please provide at least one liked movie for
recommendations.")
            return []

        # Calculate the average genre vector for liked movies to represent
user preference
        user_genre_profile = None
        for movie_title in liked_movies:
            movie_vector = self.get_movie_genre_vector(movie_title)
            if movie_vector is not None:
                if user_genre_profile is None:
                    user_genre_profile = movie_vector
                else:
                    user_genre_profile += movie_vector

        if user_genre_profile is None:
            print("Could not create user profile from liked movies. No
recommendations.")
            return []

        # Calculate similarity between user profile and all other movies
        similarities = cosine_similarity(user_genre_profile,
self.genre_matrix)

        # Get indices of movies sorted by similarity
        # Flatten similarities array and get indices in descending order
        sorted_indices = similarities.argsort()[0][::-1]

        recommendations = []
        for idx in sorted_indices:
            movie_title = self.movies_df.iloc[idx]['title']
            if movie_title not in liked_movies: # Don't recommend movies
already liked
                recommendations.append(movie_title)
            if len(recommendations) >= num_recommendations:
                break

        return recommendations

# --- Main Execution ---
if __name__ == "__main__":
    # Sample Movie Dataset
    movies_data = [
        {"title": "The Matrix", "genre": "Action, Sci-Fi", "description": "A
computer hacker learns from mysterious rebels about the true nature of his
reality and his role in the war against its controllers."},
        {"title": "Inception", "genre": "Sci-Fi, Action, Thriller",
"description": "A thief who steals corporate secrets through use of dream-
sharing technology is given the inverse task of planting an idea into the
mind of a C.E.O."},
        {"title": "Pulp Fiction", "genre": "Crime, Drama", "description":
"The lives of two mob hitmen, a boxer, a gangster's wife, and a pair of diner
bandits intertwine in four tales of violence and redemption."},
        {"title": "Forrest Gump", "genre": "Drama, Romance", "description":
"The presidencies of Kennedy and Johnson, the Vietnam War, the Watergate
```

```
scandal and other historical events unfold from the perspective of an Alabama
man with an IQ of 75."},
        {"title": "Blade Runner 2049", "genre": "Sci-Fi, Drama, Mystery",
"description": "A young blade runner's discovery of a long-buried secret
leads him to track down former blade runner Rick Deckard, who's been missing
for 30 years."},
        {"title": "Interstellar", "genre": "Sci-Fi, Drama, Adventure",
"description": "A team of explorers travel through a wormhole in space in an
attempt to ensure humanity's survival."},
        {"title": "The Shawshank Redemption", "genre": "Drama",
"description": "Two imprisoned men bond over a number of years, finding
solace and eventual redemption through acts of common decency."},
        {"title": "Spirited Away", "genre": "Animation, Adventure, Family",
"description": "During her family's move to the suburbs, a sullen 10-year-old
girl wanders into a world ruled by gods, witches, and spirits, and where
humans are changed into beasts."},
        {"title": "Arrival", "genre": "Sci-Fi, Drama, Mystery",
"description": "A linguist is recruited by the military to assist in
translating alien communications."},
        {"title": "The Lion King", "genre": "Animation, Adventure, Drama",
"description": "Lion cub Simba flees into exile after his evil uncle Scar
murders his father, Mufasa, but returns to reclaim his homeland."},
    ]

    recommender = MovieRecommender(movies_data)

    # User's liked movies
    user_liked_movies_1 = ["The Matrix", "Inception"]
    print(f"\nUser liked: {user_liked_movies_1}")
    recommendations_1 = recommender.recommend_movies(user_liked_movies_1)
    print(f"Recommended movies: {recommendations_1}")

    user_liked_movies_2 = ["Forrest Gump", "The Shawshank Redemption"]
    print(f"\nUser liked: {user_liked_movies_2}")
    recommendations_2 = recommender.recommend_movies(user_liked_movies_2)
    print(f"Recommended movies: {recommendations_2}")

    user_liked_movies_3 = ["Spirited Away"]
    print(f"\nUser liked: {user_liked_movies_3}")
    recommendations_3 = recommender.recommend_movies(user_liked_movies_3)
    print(f"Recommended movies: {recommendations_3}")

    user_liked_movies_4 = ["Non-existent Movie"]
    print(f"\nUser liked: {user_liked_movies_4}")
    recommendations_4 = recommender.recommend_movies(user_liked_movies_4)
    print(f"Recommended movies: {recommendations_4}")
```

Input

The input for this program is a list of movie titles that a user has "liked." The program then uses these liked movies to generate recommendations from its internal movie dataset.

Example Input:

- `liked_movies = ["The Matrix", "Inception"]`
- `liked_movies = ["Forrest Gump", "The Shawshank Redemption"]`
- `liked_movies = ["Spirited Away"]`

Expected Output

The program will print the movies the user liked and then a list of recommended movies based on the genre similarity to the liked movies.

Example Expected Output:

```
Movie data loaded and genre matrix created.
Available movies: ['The Matrix', 'Inception', 'Pulp Fiction', 'Forrest Gump',
'Blade Runner 2049', 'Interstellar', 'The Shawshank Redemption', 'Spirited
Away', 'Arrival', 'The Lion King']

User liked: ['The Matrix', 'Inception']
Recommended movies: ['Blade Runner 2049', 'Interstellar', 'Arrival', 'Pulp
Fiction', 'Forrest Gump']

User liked: ['Forrest Gump', 'The Shawshank Redemption']
Recommended movies: ['Pulp Fiction', 'The Matrix', 'Inception', 'Blade Runner
2049', 'Interstellar']

User liked: ['Spirited Away']
Recommended movies: ['The Lion King', 'The Matrix', 'Inception', 'Pulp
Fiction', 'Forrest Gump']

User liked: ['Non-existent Movie']
Error: Movie 'Non-existent Movie' not found.
Could not create user profile from liked movies. No recommendations.
Recommended movies: []
```

Lab 15: Write a python program for text generation using BART model.

Title

Text Generation using BART Model

Aim

To implement a Python program that utilizes a pre-trained BART (Bidirectional and Auto-Regressive Transformers) model from the Hugging Face Transformers library for abstractive text generation.

Procedure

1. **Setup Environment:**
    - Install the `transformers` and `torch` (or `tensorflow`) libraries.
2. **Load Pre-trained BART Model and Tokenizer:**
    - Import `AutoTokenizer` and `AutoModelForSeq2SeqLM` (as BART is a sequence-to-sequence model).
    - Load a pre-trained BART model (e.g., `facebook/bart-large-cnn` for summarization, or a fine-tuned BART for general generation) and its corresponding tokenizer.
3. **Prepare Input Prompt:**
    - Provide a starting prompt or a context for the text generation.
    - Tokenize and encode the input using the BART tokenizer.
    - Ensure the input is within the model's maximum sequence length.
4. **Generate Text:**
    - Pass the tokenized input to the model's `generate` method.
    - Specify generation parameters like `max_length`, `num_beams` (for beam search), `do_sample` (for diverse outputs), `top_k`, `top_p`, etc.
5. **Decode and Display:**
    - Decode the generated token IDs back into human-readable text using the tokenizer.
    - Print the input prompt and the generated text.

Source Code

```
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM
import torch

# --- Configuration ---
# BART model for conditional text generation (e.g., summarization, but can be
adapted)
# 'facebook/bart-large-cnn' is often used for summarization, but can generate
text given a prompt.
MODEL_NAME = "facebook/bart-large-cnn"

# --- 1. Setup Environment & Load Model/Tokenizer ---
print(f"Loading tokenizer and model for {MODEL_NAME}...")
try:
    tokenizer = AutoTokenizer.from_pretrained(MODEL_NAME)
    model = AutoModelForSeq2SeqLM.from_pretrained(MODEL_NAME)
    print("BART model and tokenizer loaded successfully.")
except Exception as e:
    print(f"Error loading BART model or tokenizer: {e}")
    tokenizer = None
    model = None
```

```
def generate_text_with_bart(prompt_text, max_length=100, num_beams=4,
do_sample=False, temperature=1.0):
    """
    Generates text using a pre-trained BART model based on the given prompt.
    """
    if tokenizer is None or model is None:
        return "Error: BART model or tokenizer not loaded. Cannot generate
text."

    # 2. Prepare Input Prompt
    # Encode the input prompt
    inputs = tokenizer(
        prompt_text,
        return_tensors="pt",
        max_length=1024, # BART's typical max input length
        truncation=True
    )

    # 3. Generate Text
    # Generate text using the model
    # num_beams > 1 enables beam search for better quality
    # do_sample=True enables sampling for more diverse outputs
    # top_k, top_p for controlling sampling
    generated_ids = model.generate(
        inputs["input_ids"],
        num_beams=num_beams,
        max_length=max_length,
        early_stopping=True,
        do_sample=do_sample,
        temperature=temperature if do_sample else 1.0, # Temperature only
applies if do_sample is True
        top_k=50 if do_sample else None,
        top_p=0.95 if do_sample else None
    )

    # 4. Decode and Display
    # Decode the generated token IDs back to text
    generated_text = tokenizer.decode(generated_ids[0],
skip_special_tokens=True)
    return generated_text

# --- Main Execution ---
if __name__ == "__main__":
    prompt_1 = "The quick brown fox jumps over the lazy dog. In the forest, a
new adventure began when"
    print("--- Text Generation Example 1 (Beam Search) ---")
    print(f"Prompt: '{prompt_1}'")
    generated_output_1 = generate_text_with_bart(prompt_1, max_length=150,
num_beams=5, do_sample=False)
    print(f"Generated Text:\n{generated_output_1}\n")

    prompt_2 = "Once upon a time, in a land far, far away, there lived a
brave knight who"
    print("--- Text Generation Example 2 (Sampling for Diversity) ---")
    print(f"Prompt: '{prompt_2}'")
    generated_output_2 = generate_text_with_bart(prompt_2, max_length=100,
num_beams=1, do_sample=True, temperature=0.9)
    print(f"Generated Text:\n{generated_output_2}\n")

    prompt_3 = "The scientific discovery of gravitational waves has opened up
new avenues for"
    print("--- Text Generation Example 3 (Scientific Context) ---")
    print(f"Prompt: '{prompt_3}'")
    generated_output_3 = generate_text_with_bart(prompt_3, max_length=120,
num_beams=4, do_sample=False)
```

```
    print(f"Generated Text:\n{generated_output_3}\n")
```

Input

The input for this program is the `prompt_text` string, which serves as the starting point or context for the text generation. You can also adjust `max_length`, `num_beams`, `do_sample`, and `temperature` to control the output's length, quality, and diversity.

Example Input:

- `"The quick brown fox jumps over the lazy dog. In the forest, a new adventure began when"`
- `"Once upon a time, in a land far, far away, there lived a brave knight who"`
- `"The scientific discovery of gravitational waves has opened up new avenues for"`

Expected Output

The program will print the input prompt and the text generated by the BART model. The generated text will be a continuation or expansion of the prompt, aiming to be coherent and relevant to the context.

Example Expected Output:

```
Loading tokenizer and model for facebook/bart-large-cnn...
BART model and tokenizer loaded successfully.

--- Text Generation Example 1 (Beam Search) ---
Prompt: 'The quick brown fox jumps over the lazy dog. In the forest, a new
adventure began when'
Generated Text:
The quick brown fox jumps over the lazy dog. In the forest, a new adventure
began when the fox, who had been living in the forest for many years, decided
to leave his home and explore the world. He had always been curious about the
world outside his forest, and he had heard many stories about the adventures
that lay beyond.

--- Text Generation Example 2 (Sampling for Diversity) ---
Prompt: 'Once upon a time, in a land far, far away, there lived a brave
knight who'
Generated Text:
Once upon a time, in a land far, far away, there lived a brave knight who was
known for his courage and strength. He was a man of great integrity and
honor, and he was always ready to help those in need. He was also a very kind
and compassionate man, and he always treated everyone with respect.

--- Text Generation Example 3 (Scientific Context) ---
Prompt: 'The scientific discovery of gravitational waves has opened up new
avenues for'
Generated Text:
The scientific discovery of gravitational waves has opened up new avenues for
understanding the universe. Scientists have been able to detect gravitational
waves from the collision of two black holes, and they are now working to
detect gravitational waves from other sources. This will allow them to learn
more about the universe and how it works.
```

Lab 16: Write a python program for auto texting using BART

Title

Auto-Texting (Text Completion) using BART Model

Aim

To implement a Python program that uses a pre-trained BART model to perform "auto-texting" or text completion, where the model generates the most likely continuation of an incomplete sentence or phrase.

Procedure

1. **Setup Environment:**
    - Install the `transformers` and `torch` (or `tensorflow`) libraries.
2. **Load Pre-trained BART Model and Tokenizer:**
    - Import `AutoTokenizer` and `AutoModelForSeq2SeqLM`.
    - Load a pre-trained BART model (e.g., `facebook/bart-large-cnn` or a similar model trained for language modeling/generation) and its corresponding tokenizer.
3. **Prepare Incomplete Text:**
    - Provide an incomplete sentence or phrase as input.
    - Tokenize and encode the input.
4. **Generate Completion:**
    - Pass the tokenized input to the model's `generate` method.
    - Crucially, set `num_beams` to a value greater than 1 (e.g., 5) and `do_sample=False` to enable beam search, which helps find the most probable sequence of words.
    - Specify `max_length` to control the length of the generated completion.
5. **Decode and Display:**
    - Decode the generated token IDs back into human-readable text.
    - Print the original incomplete text and its completed version.

Source Code

```
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM
import torch

# --- Configuration ---
# Using a BART model suitable for general text generation/completion.
# 'facebook/bart-large-cnn' is a good general-purpose BART model.
MODEL_NAME = "facebook/bart-large-cnn"

# --- 1. Setup Environment & Load Model/Tokenizer ---
print(f"Loading tokenizer and model for {MODEL_NAME}...")
try:
    tokenizer = AutoTokenizer.from_pretrained(MODEL_NAME)
    model = AutoModelForSeq2SeqLM.from_pretrained(MODEL_NAME)
    print("BART model and tokenizer loaded successfully.")
except Exception as e:
    print(f"Error loading BART model or tokenizer: {e}")
    tokenizer = None
    model = None

def auto_complete_text_with_bart(incomplete_text, max_length=50,
num_beams=5):
    """
    Auto-completes the given incomplete text using a pre-trained BART model.
    Uses beam search for more coherent completions.
```

```
    """
    if tokenizer is None or model is None:
        return "Error: BART model or tokenizer not loaded. Cannot auto-
complete."

    # 2. Prepare Incomplete Text
    # Encode the input text
    inputs = tokenizer(
        incomplete_text,
        return_tensors="pt",
        max_length=1024, # BART's typical max input length
        truncation=True
    )

    # 3. Generate Completion
    # Generate text using the model with beam search
    generated_ids = model.generate(
        inputs["input_ids"],
        num_beams=num_beams,
        max_length=max_length,
        early_stopping=True,
        do_sample=False # Use beam search for deterministic, most likely
completion
    )

    # 4. Decode and Display
    # Decode the generated token IDs back to text
    completed_text = tokenizer.decode(generated_ids[0],
skip_special_tokens=True)
    return completed_text

# --- Main Execution ---
if __name__ == "__main__":
    incomplete_sentence_1 = "The cat sat on the"
    print("--- Auto-Completion Example 1 ---")
    print(f"Incomplete Text: '{incomplete_sentence_1}'")
    completed_output_1 = auto_complete_text_with_bart(incomplete_sentence_1,
max_length=30)
    print(f"Completed Text:\n{completed_output_1}\n")

    incomplete_sentence_2 = "Artificial intelligence will revolutionize many
industries by"
    print("--- Auto-Completion Example 2 ---")
    print(f"Incomplete Text: '{incomplete_sentence_2}'")
    completed_output_2 = auto_complete_text_with_bart(incomplete_sentence_2,
max_length=50)
    print(f"Completed Text:\n{completed_output_2}\n")

    incomplete_sentence_3 = "In the near future, self-driving cars will"
    print("--- Auto-Completion Example 3 ---")
    print(f"Incomplete Text: '{incomplete_sentence_3}'")
    completed_output_3 = auto_complete_text_with_bart(incomplete_sentence_3,
max_length=40)
    print(f"Completed Text:\n{completed_output_3}\n")
```

Input

The input for this program is the `incomplete_text` string, which is the partial sentence or phrase that the BART model will complete. You can also adjust `max_length` and `num_beams` to control the length and quality of the completion.

Example Input:

- "The cat sat on the"
- "Artificial intelligence will revolutionize many industries by"
- "In the near future, self-driving cars will"

Expected Output

The program will print the original incomplete text and then the completed version generated by the BART model. The completion will be a coherent and grammatically plausible continuation of the input.

Example Expected Output:

```
Loading tokenizer and model for facebook/bart-large-cnn...
BART model and tokenizer loaded successfully.

--- Auto-Completion Example 1 ---
Incomplete Text: 'The cat sat on the'
Completed Text:
The cat sat on the mat.

--- Auto-Completion Example 2 ---
Incomplete Text: 'Artificial intelligence will revolutionize many industries
by'
Completed Text:
Artificial intelligence will revolutionize many industries by automating
tasks, improving efficiency, and enabling new capabilities.

--- Auto-Completion Example 3 ---
Incomplete Text: 'In the near future, self-driving cars will'
Completed Text:
In the near future, self-driving cars will be able to navigate complex urban
environments, and will be able to communicate with other vehicles and
infrastructure.
```

Lab 17: Write a python program for text summarization using BART model

Title

Text Summarization using BART Model

Aim

To implement a Python program that utilizes a pre-trained BART (Bidirectional and Auto-Regressive Transformers) model specifically for abstractive text summarization. This lab focuses on using BART's capabilities as a sequence-to-sequence model for summarization tasks.

Procedure

1. **Setup Environment:**
    - Install the `transformers` and `torch` (or `tensorflow`) libraries.
2. **Load Pre-trained BART Model and Tokenizer:**
    - Import `AutoTokenizer` and `AutoModelForSeq2SeqLM`.
    - Load a pre-trained BART model specifically fine-tuned for summarization (e.g., `facebook/bart-large-cnn`) and its corresponding tokenizer.
3. **Prepare Input Text:**
    - Provide the long text you want to summarize.
    - Tokenize and encode the input text using the BART tokenizer.
    - Ensure the input is within the model's maximum sequence length (BART models typically handle longer inputs than some other models).
4. **Generate Summary:**
    - Pass the tokenized input to the model's `generate` method.
    - Specify generation parameters crucial for summarization:
        - `num_beams` (for beam search, typically 4 or more for better quality).
        - `min_length` and `max_length` to control the summary's length.
        - `early_stopping=True` to stop generation when all beam hypotheses have reached the end-of-sentence token.
5. **Decode and Display:**
    - Decode the generated token IDs back into human-readable text, skipping special tokens.
    - Print the original text and its generated summary.

Source Code

```
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM
import torch

# --- Configuration ---
# BART model specifically fine-tuned for CNN/DailyMail summarization.
MODEL_NAME = "facebook/bart-large-cnn"

# --- 1. Setup Environment & Load Model/Tokenizer ---
print(f"Loading tokenizer and model for {MODEL_NAME}...")
try:
    tokenizer = AutoTokenizer.from_pretrained(MODEL_NAME)
    model = AutoModelForSeq2SeqLM.from_pretrained(MODEL_NAME)
    print("BART summarization model and tokenizer loaded successfully.")
except Exception as e:
    print(f"Error loading BART model or tokenizer: {e}")
    tokenizer = None
```

```
    model = None

def summarize_text_with_bart_model(text_to_summarize, min_length=50,
max_length=150, num_beams=4):
    """
    Summarizes the given text using a pre-trained BART model.
    """
    if tokenizer is None or model is None:
        return "Error: BART model or tokenizer not loaded. Cannot summarize."

    # 2. Prepare Input Text
    # Encode the input text
    inputs = tokenizer(
        text_to_summarize,
        return_tensors="pt",
        max_length=1024, # BART's typical max input length
        truncation=True
    )

    # 3. Generate Summary
    # Generate the summary using the model
    summary_ids = model.generate(
        inputs["input_ids"],
        num_beams=num_beams, # Use beam search for better quality summaries
        min_length=min_length,
        max_length=max_length,
        early_stopping=True # Stop when all beam hypotheses have reached EOS
token
    )

    # 4. Decode and Display
    # Decode the generated token IDs back to text
    summary = tokenizer.decode(summary_ids[0], skip_special_tokens=True)
    return summary

# --- Main Execution ---
if __name__ == "__main__":
    long_article_1 = """
    The Amazon rainforest is the largest rainforest in the world, covering an
immense area across nine South American countries, primarily Brazil. It is
renowned for its unparalleled biodiversity, housing an estimated 10% of the
world's known species, including countless insects, plants, birds, and
mammals. The Amazon plays a crucial role in regulating the Earth's climate by
absorbing vast amounts of carbon dioxide and releasing oxygen, often referred
to as the "lungs of the Earth." However, this vital ecosystem faces severe
threats from deforestation, primarily due to agricultural expansion, logging,
and mining. The loss of the Amazon rainforest has significant implications
for global climate change, indigenous communities, and the survival of
numerous species. Conservation efforts are underway, but the challenge
remains immense. Recent studies indicate that deforestation rates have surged
in certain areas, leading to increased concerns among environmentalists and
international bodies. Protecting the Amazon is not just a regional issue but
a global imperative for climate stability and biodiversity preservation.
    """

    long_article_2 = """
    Quantum computing is an emerging technology that harnesses the principles
of quantum mechanics to solve problems too complex for classical computers.
Unlike classical bits, which can only be 0 or 1, quantum bits (qubits) can
exist in multiple states simultaneously due to superposition. Qubits can also
be entangled, meaning their states are linked, even when physically
separated. These unique properties allow quantum computers to perform certain
calculations exponentially faster than their classical counterparts. While
still in its early stages, quantum computing holds immense promise for fields
like drug discovery, materials science, cryptography, and complex
optimization problems. However, building and maintaining stable qubits is a
```

```
significant engineering challenge, and the technology faces hurdles such as
decoherence and error correction. Research and development in this field are
accelerating globally, with major tech companies and governments investing
heavily.
    """

    print("--- Text Summarization Example 1 (BART) ---")
    print("Original Text:")
    print(long_article_1)
    summary_1 = summarize_text_with_bart_model(long_article_1, min_length=60,
max_length=120, num_beams=4)
    print("\nSummary:")
    print(summary_1)

    print("\n" + "="*50 + "\n")

    print("--- Text Summarization Example 2 (BART) ---")
    print("Original Text:")
    print(long_article_2)
    summary_2 = summarize_text_with_bart_model(long_article_2, min_length=70,
max_length=130, num_beams=4)
    print("\nSummary:")
    print(summary_2)
```

Input

The input for this program is the `text_to_summarize` string, which is the longer document you want to condense. You can also adjust `min_length`, `max_length`, and `num_beams` to control the length and quality of the generated summary.

Example Input:

- A long article about the Amazon rainforest.
- A detailed explanation of quantum computing.

Expected Output

The program will print the original long text and then its abstractive summary generated by the pre-trained BART model. The summary will be a concise and coherent condensation of the original content.

Example Expected Output:

```
Loading tokenizer and model for facebook/bart-large-cnn...
BART summarization model and tokenizer loaded successfully.

--- Text Summarization Example 1 (BART) ---
Original Text:
    The Amazon rainforest is the largest rainforest in the world, covering an
immense area across nine South American countries, primarily Brazil. It is
renowned for its unparalleled biodiversity, housing an estimated 10% of the
world's known species, including countless insects, plants, birds, and
mammals. The Amazon plays a crucial role in regulating the Earth's climate by
absorbing vast amounts of carbon dioxide and releasing oxygen, often referred
to as the "lungs of the Earth." However, this vital ecosystem faces severe
threats from deforestation, primarily due to agricultural expansion, logging,
and mining. The loss of the Amazon rainforest has significant implications
for global climate change, indigenous communities, and the survival of
numerous species. Conservation efforts are underway, but the challenge
remains immense. Recent studies indicate that deforestation rates have surged
```

in certain areas, leading to increased concerns among environmentalists and international bodies. Protecting the Amazon is not just a regional issue but a global imperative for climate stability and biodiversity preservation.

Summary:
The Amazon rainforest is the largest rainforest in the world, covering an immense area across nine South American countries. It is renowned for its unparalleled biodiversity, housing an estimated 10% of the world's known species. The Amazon plays a crucial role in regulating the Earth's climate by absorbing vast amounts of carbon dioxide and releasing oxygen, often referred to as the "lungs of the Earth." However, this vital ecosystem faces severe threats from deforestation, primarily due to agricultural expansion, logging, and mining.

==========

--- Text Summarization Example 2 (BART) ---
Original Text:
    Quantum computing is an emerging technology that harnesses the principles of quantum mechanics to solve problems too complex for classical computers. Unlike classical bits, which can only be 0 or 1, quantum bits (qubits) can exist in multiple states simultaneously due to superposition. Qubits can also be entangled, meaning their states are linked, even when physically separated. These unique properties allow quantum computers to perform certain calculations exponentially faster than their classical counterparts. While still in its early stages, quantum computing holds immense promise for fields like drug discovery, materials science, cryptography, and complex optimization problems. However, building and maintaining stable qubits is a significant engineering challenge, and the technology faces hurdles such as decoherence and error correction. Research and development in this field are accelerating globally, with major tech companies and governments investing heavily.

Summary:
Quantum computing is an emerging technology that harnesses the principles of quantum mechanics to solve problems too complex for classical computers. Unlike classical bits, which can only be 0 or 1, quantum bits (qubits) can exist in multiple states simultaneously due to superposition. Qubits can also be entangled, meaning their states are linked, even when physically separated. These unique properties allow quantum computers to perform certain calculations exponentially faster than their classical counterparts.