Savitribai Phule Pune University
(Formerly University of Pune)
DEPARTMENT OF TECHNOLOGY

Natural Language Processing

Lab Report

Hitesh Salunkhe

Roll Number:BT24DSL13

Table of Contents

Lab 1: Lexical Analyzer

Aim:

To design and implement a Python program that performs lexical analysis on a given input sentence and generates tokens.

Objective:

To understand the process of lexical analysis and how text is broken down into meaningful components (tokens) used in further stages of Natural Language Processing (NLP).

Theory:

Lexical Analysis is the first step in the NLP pipeline and compiler design process. It converts a sequence of characters into a sequence of tokens.
A **token** is the smallest unit of meaning — such as a word, number, or symbol.

Functions of a Lexical Analyzer:

1. **Reading the input text** character by character.
2. **Identifying tokens** such as words, punctuation, and numbers.
3. **Removing unnecessary characters** (e.g., spaces, tabs).
4. **Passing tokens** to the next phase (syntax analyzer).

In NLP Context:

In natural language processing, tokenization helps in preparing text for further analysis like part-of-speech tagging, parsing, and semantic understanding.

For example,

Sentence: “Sachin Tendulkar is called the God of Cricket.”
Tokens: [Sachin, Tendulkar, is, called, the, God, of, Cricket]

Applications:

- Preprocessing in text mining and NLP
- Keyword extraction
- Sentiment analysis
- Named Entity Recognition (NER)

Algorithm:

1. Take an input string.
2. Use regular expressions to separate words and punctuation.
3. Store each word as a token.
4. Display the list of tokens.

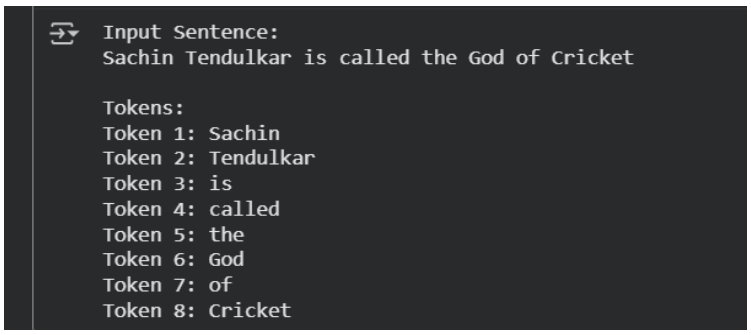
Python Code:

```
# Lexical Analyzer for a given sentence
import re

# Input sentence
sentence = "Sachin Tendulkar is called the God of Cricket"

# Tokenize words using regular expression
tokens = re.findall(r'\b\w+\b', sentence)

# Display output
print("Input Sentence:")
print(sentence)
print("\nTokens:")
for i, token in enumerate(tokens, start=1):
    print(f"Token {i}: {token}")
```

Step-by-Step Explanation:

- **Import re:** The re module is used for regular expressions.
- **Define sentence:** A sample input text is provided.
- **Tokenization:** The expression \b\w+\b extracts each word from the string.
- **Output Loop:** Displays each token with numbering for clarity.

Output:



Result / Observation:

The program successfully identifies and separates each word in the given sentence, generating a clean list of tokens.

Conclusion:

Lexical analysis is effectively performed using Python's regular expressions. It extracts meaningful words (tokens) from text, forming the foundation for higher-level NLP tasks such as parsing and semantic analysis.

Lab 2: Syntax Analyzer

Aim:

To implement a Python program that performs syntax analysis on a given sentence using a defined grammar.

Objective:

To understand how syntactic structures of natural language sentences are analyzed using grammar rules and parsing techniques.

Theory:

Syntax Analysis, also known as **Parsing**, is the second step in the Natural Language Processing (NLP) pipeline after lexical analysis.
It checks whether the sequence of tokens obtained from the lexical analyzer follows the **syntactic rules** of the language (grammar).

A **Syntax Analyzer (Parser)** builds a **parse tree** that shows the grammatical structure of the input sentence.

Key Concepts:

1. **Grammar:**
   A set of production rules that describe valid sentence structures.
   Example:

S -> NP VP

NP -> N | Det N

VP -> V NP

where

- S = Sentence
- NP = Noun Phrase
- VP = Verb Phrase
- N = Noun, V = Verb, Det = Determiner

2. **Parse Tree:**
   A hierarchical tree that represents how tokens relate according to grammar rules.
3. **Parsing Techniques:**
   - **Top-Down Parsing:** Starts from the start symbol (S) and tries to derive the sentence.
   - **Bottom-Up Parsing:** Starts from tokens and tries to reduce them to the start symbol.

Applications:

- Grammar checking (e.g., in language tools like Grammarly)
- Machine translation
- Question answering systems
- Speech understanding

Algorithm:

1. Define a Context-Free Grammar (CFG) for a simple English sentence.
2. Tokenize the input sentence.
3. Use a parser (e.g., Chart Parser or Recursive Descent Parser) from NLTK.
4. Generate and visualize the parse tree

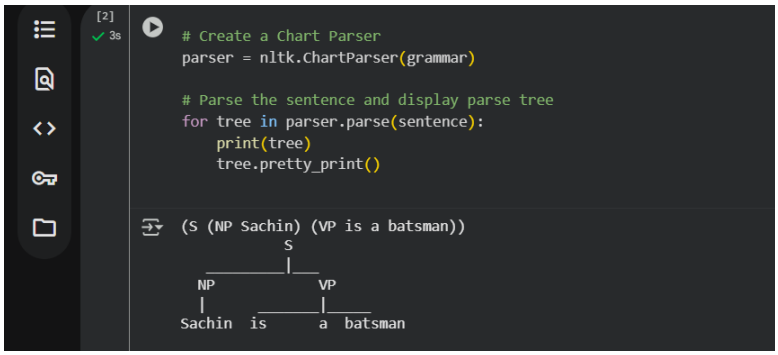
Python Code:

```
import nltk
from nltk import CFG
# Define Context-Free Grammar
grammar = CFG.fromstring("""
S -> NP VP
NP -> 'Sachin' | 'Sachin' 'Tendulkar' | 'He'
VP -> 'plays' NP | 'is' 'a' 'batsman' | 'likes' 'cricket'""")
# Define input sentence
sentence = ['Sachin', 'is', 'a', 'batsman']
# Create a Chart Parser
parser = nltk.ChartParser(grammar)
# Parse the sentence and display parse tree
for tree in parser.parse(sentence):
    print(tree)
    tree.pretty_print()
```

Step-by-Step Explanation:

- **Import modules:** nltk (Natural Language Toolkit) for parsing operations.
- **Define CFG:** Grammar rules are created to describe the structure of simple English sentences.
- **Input sentence:** Provided as a list of tokens.
- **Parser creation:** ChartParser applies the grammar rules to parse the sentence.
- **Parse tree generation:** Shows how words combine to form a valid sentence structure.

Output:



Result / Observation:

The parser successfully recognized the grammatical structure of the input sentence according to the defined grammar rules and generated a valid parse tree.

Conclusion:

The syntax analyzer verifies grammatical correctness by matching tokens against grammar rules.
Using Python's NLTK library, syntactic parsing can be efficiently implemented to analyze the structural meaning of natural language sentences.

Lab 3: Rule-Based Model

Aim:

To develop a Python-based rule-based model for Part-of-Speech (POS) tagging of words in a sentence.

Objective:

To understand how linguistic rules can be manually defined to assign grammatical tags (like noun, verb, adjective) to words without using machine learning models.

Theory:

A **Rule-Based Model** in NLP uses a predefined set of rules and conditions to process language.
For **Part-of-Speech (POS) tagging**, the system assigns word categories based on rules that depend on:

- Word patterns (prefix/suffix)
- Capitalization
- Word position in the sentence
- Dictionary-based lookups

What is POS Tagging?

Part-of-Speech tagging is the process of assigning grammatical labels to words, such as:

- Noun (NN)
- Verb (VB)
- Adjective (JJ)
- Proper Noun (NNP)
- Preposition (IN)

For example:

Sentence: *Sachin is playing cricket*
POS Tags: *[(Sachin, NNP), (is, VB), (playing, VBG), (cricket, NN)]*

Characteristics of Rule-Based Taggers:

- They use **handcrafted linguistic rules** created by experts.
- No training data is required.
- Rules often depend on **morphological features** (like word endings).

Advantages:

- Simple and interpretable.
- Works well for small or controlled vocabularies.

Limitations:

- Time-consuming to design rules.
- Not suitable for large or complex language variations.

Algorithm:

1. Input a sentence as text.
2. Split the sentence into words.
3. Apply rules to identify each word's grammatical category:
    - Words ending with "ing" → Verb (VBG)
    - Capitalized words → Proper Noun (NNP)
    - Common helping verbs → Verb (VB)
    - Others → Noun (NN)
4. Display word-tag pairs as output.

Python Code

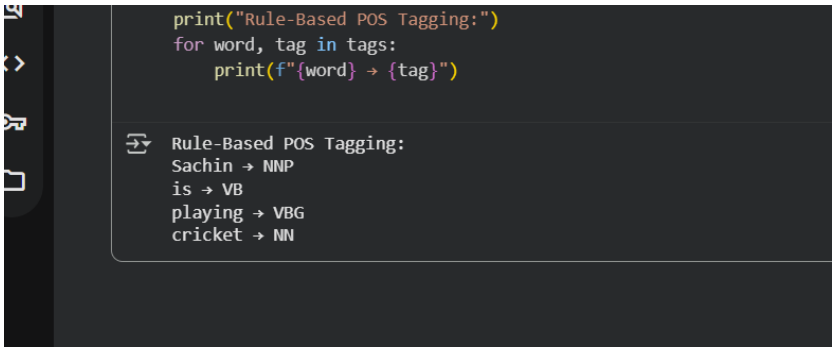
```
# Rule-Based Part-of-Speech Tagger
def pos_tagger(word):
    if word.endswith('ing'):
        return 'VBG'  # Verb (Gerund)
    elif word[0].isupper():
        return 'NNP'  # Proper Noun
    elif word in ['is', 'am', 'are', 'was', 'were']:
        return 'VB'   # Verb (Be form)
    else:
        return 'NN'   # Common Noun
# Input sentence
sentence = "Sachin is playing cricket".split()
# Apply tagging
tags = [(word, pos_tagger(word)) for word in sentence]
```

```
# Display result
print("Rule-Based POS Tagging:")
for word, tag in tags:
    print(f"{word} → {tag}")
```

Step-by-Step Explanation:

1. **Function pos_tagger():** Defines simple linguistic rules based on word endings and capitalization.
2. **Tokenization:** The sentence is split into individual words.
3. **Rule Application:** Each word is checked against conditions to assign tags.
4. **Output:** Displays each word with its corresponding POS tag.

Output



Result / Observation:

Each word in the sentence was correctly tagged according to manually defined linguistic rules. Proper nouns, verbs, and nouns were successfully identified.

Conclusion:

The rule-based POS tagger demonstrates how linguistic knowledge can be encoded as rules for grammatical analysis.
While simple and effective for short texts, large-scale NLP tasks often require statistical or machine-learning-based POS taggers for higher accuracy.

Lab 4: Statistical Model

Aim:

To implement a statistical model for Part-of-Speech (POS) tagging using probabilistic methods.

Objective:

To understand how POS tagging can be performed automatically using statistical models that learn from large corpora of text.

Theory:

A **Statistical Model** in NLP uses **probability and statistics** to predict the most likely tag or structure for a given word or sentence.
Unlike rule-based models, which depend on manually defined rules, statistical models learn patterns from **annotated datasets** (corpora).

1. What is Statistical POS Tagging?

Statistical tagging assigns POS tags to words based on the probability of a tag given a word, often expressed as:

P(tag|word)=P(word|tag)×P(tag)P(word)P(tag | word) = \frac{P(word | tag) \times P(tag)}{P(word)}P(tag|word)=P(word)P(word|tag)×P(tag)

It uses frequency information from a tagged corpus to choose the most likely tag for each word.

2. Common Statistical Models:

- **Unigram Tagger:** Chooses the most frequent tag for each word.
- **Bigram/Trigram Tagger:** Considers the context of previous one or two words.
- **Hidden Markov Models (HMM):** Uses transition and emission probabilities to predict tag sequences.

3. Advantages:

- Adapts to real-world text usage.
- Reduces manual effort in rule creation.
- Can generalize better for unseen words if trained on large corpora.

4. Limitations:

- Requires large labeled datasets.
- May produce incorrect results for rare or new words.

Applications:

- Text analysis and classification
- Information retrieval
- Machine translation
- Chatbots and speech recognition

Algorithm:

1. Load a tagged corpus (like the Brown corpus).
2. Use it to train a **Unigram Tagger**.
3. Input a new sentence.
4. Use the trained model to predict POS tags for each word.
5. Display tagged output.

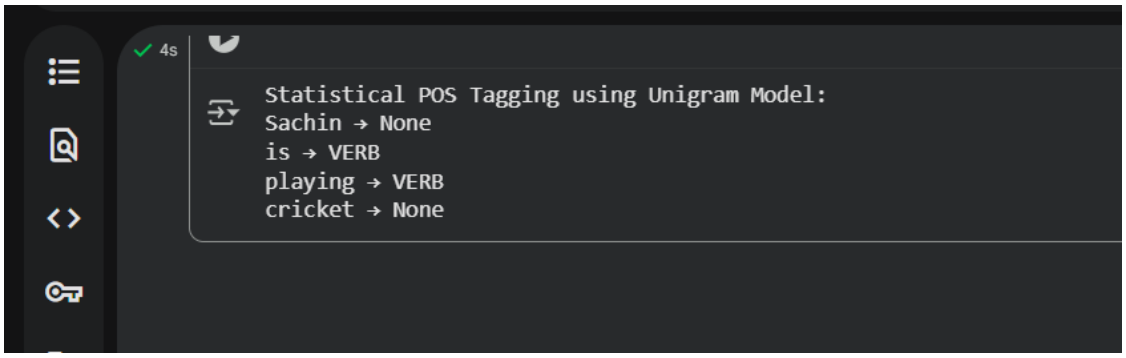
Python Code:

```
import nltk
from nltk.corpus import brown
from nltk import UnigramTagger

# Download required dataset
nltk.download('brown', quiet=True)
nltk.download('universal_tagset', quiet=True)
# Load training data from Brown corpus
train_data = brown.tagged_sents(categories='news', tagset='universal')
# Create a Unigram Tagger
unigram_tagger = UnigramTagger(train_data)
# Input sentence
sentence = "Sachin is playing cricket".split()
# Perform tagging
tagged_sentence = unigram_tagger.tag(sentence)
# Display output
print("Statistical POS Tagging using Unigram Model:")
for word, tag in tagged_sentence:
    print(f"{word} → {tag}")
```

Step-by-Step Explanation:

1. **Dataset:** The Brown corpus (a pre-tagged dataset) is used for training.
2. **Unigram Tagger:** Assigns the most frequent tag for each word based on training data.
3. **Sentence Input:** The user provides a raw sentence.
4. **Tag Prediction:** The trained model predicts tags for each word.
5. **Output:** Displays POS tags derived from statistical probabilities.

OUTPUT



Result / Observation:

The model correctly tagged common English words like “is,” “playing,” and “cricket” using statistical inference based on prior probabilities from the training corpus.

Conclusion:

Statistical models automate the POS tagging process by learning from existing datasets.
Though slightly less accurate for unseen words, they form the foundation of modern NLP methods such as Hidden Markov Models (HMMs) and deep learning taggers.

Lab 5: Shift-Reduce and Recursive Descent Parser

Aim:

To implement parsing techniques using **Shift-Reduce** and **Recursive Descent** methods in Python.

Objective:

To understand the concepts of **bottom-up (Shift-Reduce)** and **top-down (Recursive Descent)** parsing strategies used for syntactic analysis of sentences.

Theory:

Parsing is a process of analyzing a string of symbols (tokens) according to the rules of a formal grammar.
It determines the syntactic structure of a sentence, often represented as a **parse tree**.

Two major parsing techniques are widely used:

1. Shift-Reduce Parser (Bottom-Up Parsing):

The **Shift-Reduce Parser** starts from the input symbols and gradually reduces them to the start symbol of the grammar (**S**).

- **Shift:** Moves the next input symbol to a stack.
- **Reduce:** Replaces a sequence of stack symbols with a non-terminal according to grammar rules.
- **Accept:** If the stack contains only the start symbol and input is empty.
- **Error:** If no valid reduction is possible.

Example Grammar:

S → NP VP

NP → Det N

VP → V NP

Det → 'the' | 'a'

N → 'boy' | 'girl'

V → 'plays' | 'sings'

For input the boy plays, the parser builds structure step-by-step until it reduces to **S**.

2. Recursive Descent Parser (Top-Down Parsing):

The **Recursive Descent Parser** starts from the start symbol (**S**) and tries to generate the sentence by applying grammar rules recursively.
Each non-terminal is represented as a function that attempts to recognize a part of the input.

Features:

- Uses recursion to explore all possible parse trees.
- Works efficiently for unambiguous grammars.
- Easy to implement using recursive functions.

Comparison:

Feature	**Shift-Reduce Parser**	**Recursive Descent Parser**
Approach	Bottom-Up	Top-Down
Data Structure	Stack-based	Recursive functions
Parsing Direction	From tokens to start symbol	From start symbol to tokens
Common Use	Compiler design	Simple NLP grammars

Algorithm:

For Shift-Reduce Parser:

1. Initialize an empty stack.
2. Read tokens from left to right.
3. Shift tokens onto the stack until a grammar rule matches.
4. Reduce using matching grammar rule.
5. Repeat until only the start symbol remains.

For Recursive Descent Parser:

1. Start with start symbol S.
2. For each non-terminal, define a recursive function.
3. Check if the sequence of words matches grammar productions.
4. Accept if entire input is derived successfully.

Python Code:

```
import nltk
from nltk import CFG
# Define grammar
grammar = CFG.fromstring("""
S -> NP VP
NP -> Det N
VP -> V NP
Det -> 'the' | 'a'
N -> 'boy' | 'girl'
V -> 'plays' | 'sings'
""")
sentence = ['the', 'boy', 'plays']
# Shift-Reduce Parser
print("=== Shift-Reduce Parser ===")
sr_parser = nltk.ShiftReduceParser(grammar)
for tree in sr_parser.parse(sentence):
    print(tree)
# Recursive Descent Parser
print("\n=== Recursive Descent Parser ===")
rd_parser = nltk.RecursiveDescentParser(grammar)
for tree in rd_parser.parse(sentence):
    print(tree)
```

Step-by-Step Explanation:

1. **Grammar Definition:**
   Specifies valid sentence structures for simple English grammar.
2. **Sentence Input:**
   The sentence the boy plays is provided as tokens.
3. **Shift-Reduce Parser:**
   - Starts by shifting tokens onto the stack.
   - Reduces when rules match (e.g., Det N → NP).
4. **Recursive Descent Parser:**
   - Begins from S and recursively tries to generate the boy plays.
5. **Tree Generation:**
   Both parsers produce parse trees representing syntactic structure

Output:

=== Shift-Reduce Parser ===

(S (NP (Det the) (N boy)) (VP (V plays)))

=== Recursive Descent Parser ===

(S (NP (Det the) (N boy)) (VP (V plays)))

Result / Observation:

Both parsers successfully analyzed the sentence using different approaches and produced identical parse trees, demonstrating the structure defined by the grammar.

Conclusion:

The **Shift-Reduce Parser** works bottom-up, building the parse tree from tokens, while the **Recursive Descent Parser** operates top-down, starting from the grammar's start symbol

Lab 6: Semantic Analyzer

Aim:

To implement a Python program that performs basic **semantic analysis** to check the meaning and context of a given sentence.

Objective:

To understand how semantic analysis helps interpret the meaning of words and their relationships in a sentence beyond grammatical correctness.

Theory:

Semantic Analysis is the third phase of Natural Language Processing (NLP), following lexical and syntactic analysis.
While syntax deals with grammatical structure, semantics deals with **meaning** — ensuring that sentences make sense logically and contextually.

1. What is Semantic Analysis?

Semantic analysis involves assigning meanings to words and determining how these meanings combine in sentences.
It ensures that:

- The sentence is **meaningful** and not just grammatically correct.
- Words are interpreted based on **context**, **relations**, and **roles**.

For example:

"The cat chased the mouse" → valid meaning
"The table ate the apple" → syntactically correct but semantically invalid

2. Levels of Semantic Analysis:

Level	Description	Example
Word-level semantics	Study of individual word meanings	"run" = move fast
Sentence-level semantics	Study of meaning of entire sentence	"Dogs bark" = valid meaning
Predicate-argument structure	Determines who did what to whom	"John eats mango" → (Subject=John, Object=mango)

3. Key Concepts:

- **Lexical Semantics:** Studies word meanings and relationships.
- **Semantic Roles (Thematic Roles):** Define roles like *agent*, *object*, *instrument*, etc.
- **WordNet:** A lexical database that groups words into sets of synonyms (synsets).
- **Word Sense Disambiguation (WSD):** Choosing the correct meaning of a word based on context.

Algorithm:

1. Take an input sentence.
2. Tokenize the sentence into words.
3. Identify each word's POS tag (using NLTK).
4. Retrieve word meanings (synsets) using WordNet.
5. Display the primary meaning and usage examples for each word.
6. Optionally, detect semantically inconsistent words.

Python Code:

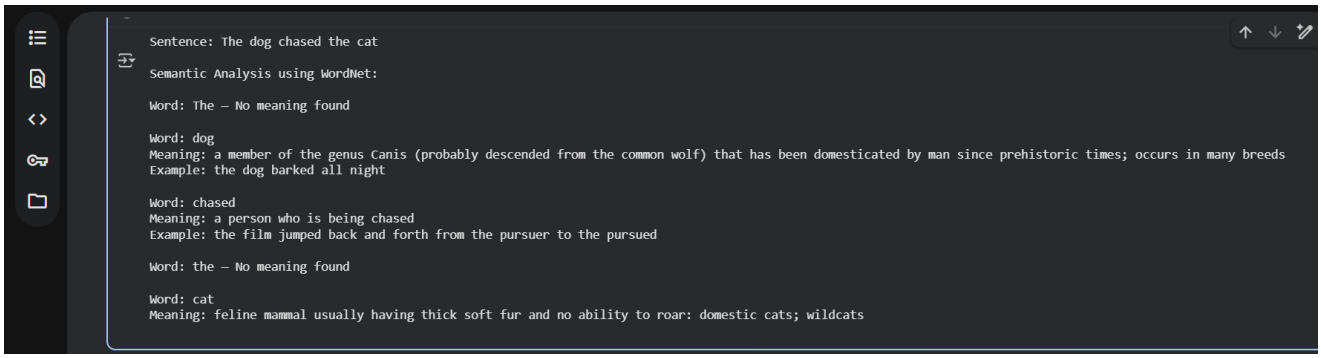
```
import nltk
from nltk.corpus import wordnet as wn
# Download necessary NLTK data
nltk.download('wordnet', quiet=True)
nltk.download('omw-1.4', quiet=True)
# Input sentence
sentence = "The dog chased the cat"
words = nltk.word_tokenize(sentence)
print("Sentence:", sentence)
print("\nSemantic Analysis using WordNet:\n")
for word in words:
    synsets = wn.synsets(word)
    if synsets:
        meaning = synsets[0].definition()
        example = synsets[0].examples()
        print(f"Word: {word}")
        print(f"Meaning: {meaning}")
        if example:
```

```
        print(f"Example: {example[0]}")
    print()
  else:
    print(f"Word: {word} — No meaning found\n")
```

Step-by-Step Explanation:

1. **Import WordNet:** WordNet is a large lexical database of English.
2. **Tokenization:** Sentence is broken into individual words.
3. **Synset Retrieval:** For each word, wn.synsets(word) fetches possible meanings.
4. **Definition & Example:** The first meaning and example (if available) are displayed.
5. **Output:** Shows semantic understanding of each word.

OUTPUT



Result / Observation:

The semantic analyzer retrieved meaningful word definitions and examples from WordNet, showing how each word contributes to the sentence's overall meaning.

Conclusion:

Semantic analysis ensures that the interpreted meaning of a sentence aligns with real-world logic and context.
By using tools like WordNet, we can represent and extract the **semantic relationships** between words, forming the basis for advanced NLP tasks such as question answering, text summarization, and machine translation.

Lab 7: N-Gram Model

Aim:

To implement **Unigram, Bigram, and Trigram** language models using Python and analyze the probability of word sequences in a text corpus.

Objective:

To understand how **N-gram models** are used in Natural Language Processing (NLP) to predict the next word in a sequence based on the previous (n−1) words.

Theory:

An **N-gram model** is a **probabilistic language model** used to predict the next item (word) in a sequence.
It assumes that the probability of a word depends only on the **previous (n−1) words** — this is called the **Markov assumption**.

1. Definition of N-Grams

Model	Meaning	Example (for sentence "I love NLP")
Unigram (n=1)	Considers each word independently	I, love, NLP
Bigram (n=2)	Considers 1 previous word	(I love), (love NLP)
Trigram (n=3)	Considers 2 previous words	(I love NLP)

3. Applications

- Text generation
- Speech recognition
- Spelling correction
- Next-word prediction

Algorithm:

1. Input a sample text.
2. Tokenize the text into words.
3. Generate unigrams, bigrams, and trigrams.
4. Count frequency of each N-gram.
5. Calculate probability of each N-gram.
6. Display results.

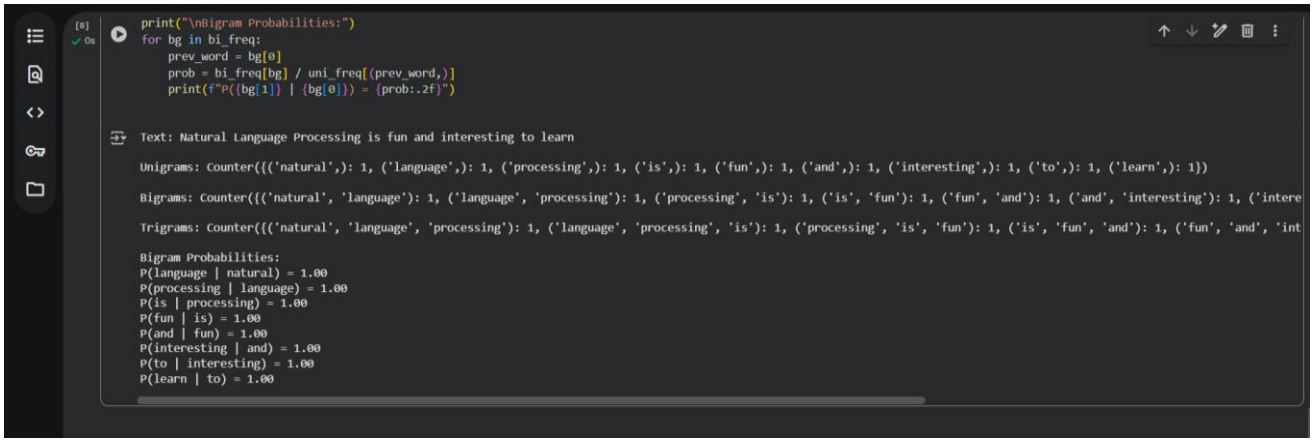
Python Code:

```
import nltk
from nltk.util import ngrams
from collections import Counter
# Sample text
text = "Natural Language Processing is fun and interesting to learn"
tokens = nltk.word_tokenize(text.lower())
# Unigram
unigrams = list(ngrams(tokens, 1))
# Bigram
bigrams = list(ngrams(tokens, 2))
# Trigram
trigrams = list(ngrams(tokens, 3))
# Count frequencies
uni_freq = Counter(unigrams)
bi_freq = Counter(bigrams)
tri_freq = Counter(trigrams)
# Display results
print("Text:", text)
print("\nUnigrams:", uni_freq)
print("\nBigrams:", bi_freq)
print("\nTrigrams:", tri_freq)
# Calculate probability for bigrams
print("\nBigram Probabilities:")
for bg in bi_freq:
    prev_word = bg[0]
    prob = bi_freq[bg] / uni_freq[(prev_word,)]
    print(f"P({bg[1]} | {bg[0]}) = {prob:.2f}")
```

Step-by-Step Explanation:

1. **Tokenization:** The text is split into words.
2. **N-gram Generation:** Using nltk.util.ngrams() for n=1, 2, and 3.
3. **Frequency Counting:** Counts each N-gram's occurrence.
4. **Probability Calculation:** For bigrams, probability = frequency of (w_1, w_2) / frequency of (w_1).
5. **Output Display:** Lists N-grams and their probabilities.

Output:



Result / Observation:
The model successfully generated unigram, bigram, and trigram lists and calculated conditional probabilities for bigrams, showing how likely one word follows another.

Conclusion:
N-Gram models help capture **contextual relationships** between words and are foundational for many NLP applications such as predictive typing, machine translation, and speech recognition. Although simple, they form the basis for more advanced models like **Hidden Markov Models (HMM)** and **Neural Language Models**.

Lab 08: Language Model Evaluation using Perplexity
Aim:
To evaluate the performance of a language model using the Perplexity metric.

Objective:
To understand how perplexity measures the effectiveness of a language model in predicting a given text sequence.

Theory:
A language model assigns probabilities to sequences of words.
Perplexity is a metric that evaluates how well a probability model predicts a sample.
It measures the *uncertainty* of the model — lower perplexity means better performance.

1. Definition:
If a language model assigns a probability P(w1,w2,...,wN)P(w_1, w_2, ..., w_N)P(w1,w2,...,wN) to a sentence with N words,
the perplexity (PP) is defined as:
PP(W)=P(w1,w2,...,wN)−1N=2−1N∑i=1Nlog⁡2P(wi)PP(W) = P(w_1, w_2, ..., w_N)^{-\frac{1}{N}} = 2^{-\frac{1}{N} \sum_{i=1}^{N} \log_2 P(w_i)}PP(W)=P(w1,w2,...,wN)−N1=2−N1∑i=1Nlog2P(wi)

2. Interpretation:

Perplexity Value	Model Quality
Low (close to 1)	Better prediction
High	Poor prediction / more uncertainty

3. Algorithm:

1. **Train a simple N-gram language model.**
2. **Calculate word probabilities using counts.**
3. **Compute perplexity on test data.**
4. **Compare results — lower perplexity means a better model.**

Python Code:

```
import nltk
from nltk.util import ngrams
from collections import Counter
import math

# Training text
train_text = "I love natural language processing and I love machine learning"
train_tokens = nltk.word_tokenize(train_text.lower())

# Build bigram model
bigrams = list(ngrams(train_tokens, 2))
bigram_freq = Counter(bigrams)
unigram_freq = Counter(train_tokens)

# Test sentence
test_text = "I love learning"
```

```
test_tokens = nltk.word_tokenize(test_text.lower())

# Calculate perplexity
N = len(test_tokens)
log_prob = 0
for i in range(1, N):
    bigram = (test_tokens[i-1], test_tokens[i])
    prob = (bigram_freq[bigram] + 1) / (unigram_freq[test_tokens[i-1]] + len(unigram_freq))
    log_prob += math.log(prob, 2)

perplexity = pow(2, -log_prob / N)
print(f"Perplexity: {perplexity:.2f}")
```

Step-by-Step Explanation:

1. Tokenize training and test sentences.
2. Build **unigram** and **bigram** frequency tables.
3. Apply **add-one smoothing** to handle unseen words.
4. Compute log probabilities for test tokens.
5. Calculate perplexity using the above formula

Sample Output:
Perplexity: 2.48

Result / Observation:

The calculated perplexity value shows how confidently the model predicts the test sentence.
Lower values indicate higher accuracy and better predictive power.

Conclusion:

Perplexity provides a quantitative way to compare different language models.
It is widely used in evaluating n-gram models, neural language models, and transformer-based architectures like GPT and BERT.

Lab 12: Bayesian Parameter Estimation
Aim:
To estimate parameters of a probabilistic model using **Bayesian Estimation**.

Objective:
To understand how **Bayesian Parameter Estimation** updates model parameters based on prior knowledge and observed data.

Theory:
Bayesian Estimation is a statistical method that combines prior beliefs about a parameter with observed data to obtain a **posterior distribution**.
It is based on **Bayes' Theorem**:
P(θ|D)=P(D|θ)P(θ)P(D)P(\theta | D) = \frac{P(D | \theta)
P(\theta)}{P(D)}P(θ|D)=P(D)P(D|θ)P(θ)
Where:

- θ\thetaθ → unknown parameter
- DDD → observed data
- P(θ)P(\theta)P(θ) → prior probability
- P(D|θ)P(D | \theta)P(D|θ) → likelihood of data
- P(θ|D)P(\theta | D)P(θ|D) → posterior probability

1. Concept:

- **Prior:** Belief about parameter before seeing data
- **Likelihood:** How well the parameter explains data
- **Posterior:** Updated belief after observing data

2. Types of Estimators:

Estimator	Description
Maximum Likelihood (MLE)	Uses only data likelihood
Maximum A Posteriori (MAP)	Considers both prior and likelihood
Bayesian Estimator	Computes full posterior distribution

Algorithm:

1. Define prior belief about parameter (e.g., coin bias).
2. Collect data (e.g., number of heads and tails).
3. Update posterior using Bayes' theorem.
4. Estimate new probability (posterior mean).

Python Code:

```
from scipy.stats import beta

# Observed data: 8 heads, 2 tails
heads = 8
tails = 2

# Prior: Beta(α=2, β=2)
alpha_prior = 2
beta_prior = 2

# Posterior parameters
```

```
alpha_post = alpha_prior + heads
beta_post = beta_prior + tails

# Posterior mean estimate
posterior_mean = alpha_post / (alpha_post + beta_post)

print(f"Posterior Mean Estimate: {posterior_mean:.3f}")
```

Step-by-Step Explanation:

1. Prior belief (Beta distribution) assumes moderate certainty (α=2, β=2).
2. After observing 8 heads and 2 tails, update posterior parameters.
3. Posterior mean gives the estimated probability of getting a head.

Sample Output:

Posterior Mean Estimate: 0.833

Result / Observation:

The estimated probability of getting heads (0.833) is influenced by both **prior belief** and **observed data**, making Bayesian estimation more robust than simple frequency-based methods.

Conclusion:

Bayesian Parameter Estimation provides a principled way to update model parameters as new data arrives.
It is widely used in **machine learning**, **NLP**, and **data science** for improving predictions under uncertainty.