### 1. ****Definition (1 Mark):****

* **Statistical Estimation for Language Models** involves the process of determining the probabilities of word sequences based on observed data, often using statistical methods to capture the language patterns present in a given corpus.

### 2. ****Key Concepts (1 Mark):****

* **N-gram Models:** Utilize the conditional probabilities of word sequences given the context of the previous (n-1) words.
* **Maximum Likelihood Estimation (MLE):** Estimate probabilities based on the observed frequencies of n-grams in the training data.
* **Smoothing Techniques:** Address the issue of zero probabilities for unseen n-grams in the training data.

### 3. ****Equations (1 Mark):****

* **MLE Formula for Bigrams:** P(wi​∣wi−1​)=count(wi−1​,wi​)​/count(wi−1​)

### 5. ****Algorithmic Steps (1 Mark):****

* **Calculate Maximum Likelihood Estimates:** Count the occurrences of n-grams in the training corpus and compute MLE probabilities.
* **Smoothing:** Apply smoothing techniques to handle unseen n-grams and improve model generalization.
* **Back-off and Interpolation:** Incorporate back-off and interpolation methods for more robust modeling.

### 6. ****Applications (1 Mark):****

* **Speech Recognition:** Language models assist in decoding speech by estimating the likelihood of word sequences.
* **Machine Translation:** Improve translation accuracy by incorporating language models into translation systems.

### 7. ****Recent Trends (1 Mark):****

* **Neural Language Models:** The rise of neural network-based language models, such as LSTM and transformers, offering improved contextual understanding.
* **Subword Models:** Leveraging subword units for handling rare and out-of-vocabulary words, improving language model coverage.

### 8. ****Pseudo Code:****

# Pseudo code for bigram MLE estimation and Laplace smoothing

def calculate\_bigram\_probabilities(corpus):

bigram\_counts = count\_bigrams(corpus)

unigram\_counts = count\_unigrams(corpus)

bigram\_probabilities = {}

smoothing\_factor = 1 # Laplace smoothing

for bigram in bigram\_counts:

context = bigram[0]

probability = (bigram\_counts[bigram] + smoothing\_factor) / (unigram\_counts[context] + smoothing\_factor \* vocabulary\_size)

bigram\_probabilities[bigram] = probability

return bigram\_probabilities

def count\_bigrams(corpus):

# Implement logic to count bigrams in the corpus

pass

def count\_unigrams(corpus):

# Implement logic to count unigrams in the corpus

pass

### 9. ****Tools (1 Mark):****

* **NLTK (Natural Language Toolkit):** Provides tools for n-gram models, MLE estimation, and various smoothing techniques.
* **KenLM:** A toolkit for building and querying efficient language models.

### 1. ****Definition (1 Mark):****

* **Parts of Speech (POS) Tagging** is a natural language processing task that involves assigning a grammatical category (such as noun, verb, adjective, etc.) to each word in a given text

### 2. ****Key Concepts (1 Mark):****

* **POS Tags:** A set of predefined tags representing different parts of speech (e.g., NN for noun, VB for verb).
* **Tokenization:** Breaking down a text into individual words or tokens.
* **Markov Models:** Statistical models that assume the probability of a particular state (POS tag) depends only on the previous state.

### 4. ****Algorithmic Steps (1 Mark):****

* **Tokenization:** Split the text into words or tokens.
* **Training:** Use annotated training data to estimate transition probabilities in an HMM or other statistical models.
* **Decoding:** Apply the Viterbi algorithm to find the most likely sequence of POS tags given the observed words.
* **CRF (Conditional Random Fields):** Another approach for sequence labeling that considers dependencies between adjacent labels.

### 5. ****Applications (1 Mark):****

* **Information Retrieval:** Enhancing search queries by understanding the syntactic structure.
* **Named Entity Recognition (NER):** Identifying entities like names, locations, and organizations in a text.

### 6. ****Recent Trends (1 Mark):****

* **Deep Learning:** Adoption of neural network architectures, such as recurrent neural networks (RNNs) and transformers, for improved sequence labeling accuracy.
* **BERT and Transformer-based Models:** Pre-trained language models that excel in capturing contextual information, benefiting POS tagging and related tasks.

### 7. ****Pseudo Code:****

python

# Pseudo code for POS tagging using NLTK

import nltk

def pos\_tagging(text):

tokens = nltk.word\_tokenize(text)

pos\_tags = nltk.pos\_tag(tokens)

return pos\_tags

# Example Usage

text = "Natural language processing is fascinating."

pos\_tags = pos\_tagging(text)

print(pos\_tags)

### 8. ****Tools (1 Mark):****

* **NLTK (Natural Language Toolkit):** A comprehensive library for natural language processing in Python, including tools for POS tagging.
* **spaCy:** An open-source NLP library that provides efficient and accurate POS tagging.
* **Stanford POS Tagger:** A part of the Stanford NLP toolkit, offering robust POS tagging.

### ****9. Sequence Labeling for Named Entity Recognition (NER):****

* **Definition (1 Mark):**
  + **Named Entity Recognition (NER)** is a type of sequence labeling that involves identifying and classifying entities (such as names, locations, and organizations) in a text.
* **Key Concepts (1 Mark):**
  + **Entity Types:** Categories of entities to be recognized (e.g., person, organization, date).
  + **BIO Encoding:** A common encoding scheme for representing entity boundaries in a sequence (B - beginning, I - inside, O - outside).
* **Applications (1 Mark):**
  + **Information Extraction:** Extracting structured information from unstructured text.
  + **Question Answering:** Improving the accuracy of answers by identifying relevant entities.
* **Recent Trends (1 Mark):**
  + **BERT and Transformer-based Models:** Pre-trained language models that have shown remarkable performance in NER tasks.
  + **Domain-Specific NER:** Tailoring NER models for specific domains to enhance accuracy in specialized contexts.
* **Pseudo Code (1 Mark):**

python

* # Pseudo code for NER using spaCy

import spacy

def ner\_extraction(text):

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

doc = nlp(text)

entities = [(ent.text, ent.label\_) for ent in doc.ents]

return entities

# Example Usage

text = "Apple Inc. was founded by Steve Jobs in Cupertino."

entities = ner\_extraction(text)

print(entities)

* **Tools (1 Mark):**
  + **spaCy:** Known for its efficiency in NER and other NLP tasks.
  + **Stanford NER:** A part of the Stanford NLP toolkit, providing a robust NER system.

### ****1. Definition (1 Mark):****

* **Lexical Syntax** refers to the set of rules governing the formation and structure of lexical elements or tokens in a programming language. It defines how characters and symbols are combined to create valid words, identifiers, and literals.

### 2. Key Concepts (1 Mark):

* **Lexeme:** The smallest unit in the lexical syntax, representing a meaningful element such as a keyword, identifier, or literal.
* **Tokenization:** The process of breaking a source code into tokens based on lexical syntax rules.
* **Regular Expressions:** Often used to describe lexical patterns and define token structures.

### 4. Algorithmic Steps (1 Mark):

* **Lexical Analysis:** Process the source code to identify and categorize lexical elements.
* **Tokenization:** Recognize and classify lexemes into specific token types.
* **Error Handling:** Identify and handle lexical errors, such as illegal characters or invalid tokens.

### 5. Applications (1 Mark):

* **Compiler Construction:** Lexical analysis is the first phase in compiling source code into machine code.
* **Syntax Highlighting:** In text editors, highlighting keywords, identifiers, and literals based on lexical rules.

### 6. Recent Trends (1 Mark):

* **Language Server Protocols:** Standard protocols for providing language services (including lexical analysis) in modern code editors.
* **Incremental Lexing:** Techniques that allow for efficient re-lexing of code segments during interactive editing.

### 7. Pseudo Code:

python

# Pseudo code for a simple lexical analyzer

def lexical\_analyzer(source\_code):

tokens = []

current\_token = ""

for char in source\_code:

if char.isalnum():

current\_token += char

else:

if current\_token:

tokens.append(classify\_token(current\_token))

current\_token = ""

if char.isspace():

continue

tokens.append(classify\_token(char))

return tokens

def classify\_token(token):

# Implement logic to classify the token into specific types (keyword, identifier, literal, etc.)

pass

# Example Usage

source\_code = "int main() { return 0; }"

tokens = lexical\_analyzer(source\_code)

print(tokens)

### 8. Tools (1 Mark):

* **Lex**: A popular lexical analyzer generator that generates lexical analyzers based on regular expressions.
* **ANTLR (ANother Tool for Language Recognition)**: A powerful parser generator that can be used for lexical analysis.

### 9. Lexical Syntax in Programming Languages (1 Mark):

* Each programming language has its lexical syntax rules defining keywords, identifiers, literals, operators, and other lexical elements.
* Examples include the lexical rules of Python, Java, C++, etc.

**Hidden Markov Models (HMMs):**

### 1. Definition (1 Mark):

* **Hidden Markov Models (HMMs)** are statistical models used to describe sequences of observable events generated by an underlying stochastic process with hidden states. They are widely applied in various fields, including speech recognition, bioinformatics, and natural language processing.

### 2. Key Concepts (1 Mark):

* **States:** Observable and hidden states represent different aspects of the system.
* **Observations:** Emissions associated with each state.
* **Transitions:** Probabilities of moving between states.

### 3. Diagram:

* **HMM Transition Diagram:** Illustrate a diagram showing states, transitions, and observations in an HMM.

### 4. Algorithmic Steps (1 Mark):

#### Forward Algorithm:

* **Definition (1 Mark):**
  + The **Forward Algorithm** calculates the probability of observing a given sequence of observations, considering all possible paths through the model.
* **Steps (2 Marks):**
  + **Initialization:** Initialize the forward probabilities for the initial state.
  + **Recursion:** Iterate through each time step, updating the forward probabilities based on the previous probabilities and transitions.
  + **Termination:** Sum the forward probabilities across all possible paths to get the overall probability of the observed sequence.

#### Viterbi Algorithm:

* **Definition (1 Mark):**
  + The **Viterbi Algorithm** is a dynamic programming algorithm that finds the most likely sequence of hidden states given a sequence of observations.
* **Steps (2 Marks):**
  + **Initialization:** Initialize the Viterbi probabilities for the initial state.
  + **Recursion:** Iterate through each time step, updating the Viterbi probabilities based on the previous probabilities, transitions, and emissions.
  + **Backtracking:** Once the sequence is observed, backtrack to find the most likely path of hidden states.

#### EM Training (Expectation-Maximization):

* **Definition (1 Mark):**
  + The **Expectation-Maximization (EM) Algorithm** is used to estimate the parameters (transition probabilities, emission probabilities) of an HMM when the hidden states are not directly observable.
* **Steps (2 Marks):**
  + **Initialization:** Start with initial estimates for the parameters.
  + **Expectation (E-step):** Calculate the expected value of the hidden states given the observed data using the current parameter estimates.
  + **Maximization (M-step):** Update the parameters to maximize the likelihood of the observed data, based on the expected values obtained in the E-step.
  + **Repeat:** Iteratively perform the E-step and M-step until convergence.

### 5. Applications (1 Mark):

* **Speech Recognition:** HMMs model the sequence of phonemes in speech.
* **Bioinformatics:** Used in gene prediction and sequence alignment.
* **Natural Language Processing:** Applied in part-of-speech tagging and named entity recognition.

### 6. Recent Trends (1 Mark):

* **Deep Learning:** Increasing use of neural networks, such as Long Short-Term Memory (LSTM) networks, for sequence modeling tasks, impacting traditional HMM applications.
* **Hybrid Models:** Combining HMMs with neural networks for improved performance in various applications.

### 7. Pseudo Code (1 Mark):

* **Forward Algorithm Pseudo Code:**

python

* def forward\_algorithm(observations, states, initial\_probs, transition\_probs, emission\_probs):

# Implement forward algorithm logic

pass

* **Viterbi Algorithm Pseudo Code:**

python

* def viterbi\_algorithm(observations, states, initial\_probs, transition\_probs, emission\_probs):

# Implement Viterbi algorithm logic

pass

* **EM Training Pseudo Code:**

python

* def em\_training(observations, states, initial\_probs, transition\_probs, emission\_probs):

# Implement EM training algorithm logic

pass

### 8. Tools (1 Mark):

* **hmmlearn:** A Python library for HMMs, providing implementations of the Forward, Viterbi algorithms, and EM training.
* **SpeechRecognition:** An open-source library for speech recognition that uses HMMs.