1.Importing libraries:

```
import re # re is for cleaning text using regular expressions
import math # math is used for perplexity  and probability calculations
from collections import Counter, defaultdict #to count word frequencies
import nltk #for tokenization and NLP utilities
nltk.download('punkt')
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
```

load dataset:

```
corpus = """
Natural language processing is a field of artificial intelligence.
It helps computers understand human language.
Language models predict the next word in a sentence.
They are widely used in chatbots and search engines.
"""
```

```
print(corpus)
```

```
Natural language processing is a field of artificial intelligence.
It helps computers understand human language.
Language models predict the next word in a sentence.
They are widely used in chatbots and search engines.
```

Preprocess text:

```
import re # re is for cleaning text using regular expressions
import nltk #for tokenization and NLP utilities
nltk.download('punkt')
nltk.download('punkt_tab') # Added to download the missing resource
from nltk.tokenize import word_tokenize

def preprocess(text):
    text = text.lower()
    text = re.sub(r'[^a-z\s]', '', text)
    tokens = word_tokenize(text)
    tokens = ['<s>'] + tokens + ['</s>']
    return tokens

tokens = preprocess(corpus)
print(tokens)
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package punkt_tab to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt_tab.zip.
['<s>', 'natural', 'language', 'processing', 'is', 'a', 'field', 'of', 'artificial', 'intelligence', 'it', 'helps', 'compute
```

Building N-gram models:

Unigram:

```
from collections import Counter # Import Counter
unigram_counts = Counter(tokens)
total_words = sum(unigram_counts.values())
```

```
print('Unigram Counts:', unigram_counts)
print('Total Words:', total_words)
```

```
Unigram Counts: Counter({'language': 3, 'a': 2, 'in': 2, '<s>': 1, 'natural': 1, 'processing': 1, 'is': 1, 'field': 1, 'of':
Total Words: 35
```

Biagram:

```
bigrams = list(zip(tokens[:-1], tokens[1:]))
bigram_counts = Counter(bigrams)

print('Bigram Counts:', bigram_counts)
print('Total Words:', total_words)
```

```
Bigram Counts: Counter({('<s>', 'natural'): 1, ('natural', 'language'): 1, ('language', 'processing'): 1, ('processing', 'is
Total Words: 35
```

Trigram:

```
trigrams = list(zip(tokens[:-2], tokens[1:-1], tokens[2:]))
trigram_counts = Counter(trigrams)
print('trigram Counts:', trigram_counts)
print('Total Words:', total_words)
```

```
trigram Counts: Counter({('<s>', 'natural', 'language'): 1, ('natural', 'language', 'processing'): 1, ('language', 'processi
Total Words: 35
```

Conditional probabilities:

Unigram probability:

```
word_to_check = 'language'
probability = unigram_prob(word_to_check)
print(f"The unigram probability of '{word_to_check}' is: {probability}")
```

```
The unigram probability of 'language' is: 0.08571428571428572
```

Biagram probability:

```
word1 = 'natural'
word2 = 'language'

# Ensure the words exist in the unigram counts to avoid division by zero for the denominator
if unigram_counts[word1] > 0:
    bigram_probability = bigram_prob(word1, word2)
    print(f"The bigram probability of '{word1} {word2}' is: {bigram_probability}")
else:
    print(f"The word '{word1}' does not appear in the corpus, so its bigram probability cannot be calculated in this model.
```

```
The bigram probability of 'natural language' is: 1.0
```

Trigram probability:

```
word1 = 'natural'
word2 = 'language'
word3 = 'processing'

# Ensure the bigram (w1, w2) exists in the bigram counts to avoid division by zero
if bigram_counts[(word1, word2)] > 0:
    trigram_probability = trigram_prob(word1, word2, word3)
    print(f"The trigram probability of '{word1} {word2} {word3}' is: {trigram_probability}")
else:
    print(f"The bigram '{word1} {word2}' does not appear in the corpus, so its trigram probability cannot be calculated in t
```

```
The trigram probability of 'natural language processing' is: 1.0
```

Smoothing:

```
V = len(unigram_counts) # Vocabulary size

def smoothed_bigram_prob(w1, w2):
    # Add-one smoothing formula for bigram probability
    return (bigram_counts[(w1, w2)] + 1) / (unigram_counts[w1] + V)
```

Sentence probability:

```
sentences = [
    "language models predict words",
    "computers understand language",
    "chatbots use language models",
    "artificial intelligence is useful",
    "language is powerful"
]
```

```
def sentence_prob(sentence):
    words = preprocess(sentence)
    prob = 1
    for i in range(len(words)-1):
        prob *= smoothed_bigram_prob(words[i], words[i+1])
    return prob
```

```
for s in sentences:
    print(s, ":", sentence_prob(s))
```

```
language models predict words : 1.1581623576850094e-07
computers understand language : 1.7951516544117647e-06
chatbots use language models : 5.790811788425047e-08
artificial intelligence is useful : 6.152737525201613e-08
language is powerful : 9.265298861480075e-07
```

Perplexity:

```
import math
def perplexity(sentence):
    words = preprocess(sentence)
    N = len(words)
    prob = sentence_prob(sentence)
    return math.pow(1/prob, 1/N)
```

```
import math
for s in sentences:
    print(s, "Perplexity:", perplexity(s))
```

```
language models predict words Perplexity: 14.323146228059127
computers understand language Perplexity: 14.098722782658538
chatbots use language models Perplexity: 16.077188053381917
artificial intelligence is useful Perplexity: 15.915560405450568
language is powerful Perplexity: 16.092670291663893
```

Comparison and Analysis:

Trigram model generally gives the lowest perplexity because it uses more context. Bigram performs better than unigram as it considers previous word. Unigram ignores word order and performs poorly. When unseen words appear, probabilities become zero without smoothing. Smoothing ensures model still gives reasonable output. Trigrams may fail if training data is very small. Bigram is a good balance between accuracy and data requirement. Smoothing improves performance for rare and unknown words.

LAB REPORT:

Objective:

The main objective of this experiment is to implement and analyze N-gram based language models and evaluate their performance using sentence probability and perplexity measures. This experiment aims to help understand how statistical language models work by predicting the likelihood of word sequences in a given text corpus. The study focuses on building Unigram, Bigram, and Trigram models and comparing their effectiveness. Another important goal is to observe the effect of smoothing techniques on model performance and understand the challenges faced by language models when encountering unseen data. This experiment also provides practical exposure to real-world Natural Language Processing (NLP) techniques used in modern applications such as chatbots, speech recognition systems, and machine translation.

Dataset Description:

The dataset used for this experiment consists of a small English text corpus related to concepts in Natural Language Processing. The corpus contains multiple sentences describing topics such as artificial intelligence, language models, and their applications. The dataset is simple and easy to understand, making it suitable for academic demonstration of N-gram models. The text data was stored in a Word document and later loaded into the system using Python libraries. This dataset helps in learning how language models are trained using real textual data. Although the dataset is limited in size, it is sufficient to illustrate how word probabilities and sequence modeling work. Using a small dataset also highlights the importance of smoothing and data size in building effective language models.

Preprocessing:

Preprocessing is a crucial step in building any NLP system, as raw text often contains noise and inconsistencies. In this experiment, the preprocessing stage involved converting all text to lowercase so that words like “Language” and “language” are treated as the same token. Punctuation marks and numerical values were removed to avoid unnecessary symbols affecting the model. The cleaned text was then tokenized, meaning the text was split into individual words or tokens. Start and end tokens were added to mark sentence boundaries, which helps the model understand where sentences begin and end. These preprocessing steps ensure that the text is in a standardized format suitable for building statistical models. Proper preprocessing significantly improves the accuracy and reliability of the N-gram models.

N-Gram Model Construction:

In this experiment, three types of N-gram models were constructed: Unigram, Bigram, and Trigram models. The Unigram model considers each word independently and calculates the probability of a word based solely on its frequency in the corpus. The Bigram model calculates the probability of a word based on the previous word, thereby capturing limited context. The Trigram model further extends this by considering the previous two words, providing a richer contextual understanding.

Word counts were computed using frequency counters, and probabilities were calculated using standard probability formulas. Conditional probabilities were used for Bigram and Trigram models. Additionally, Add-One (Laplace) smoothing was applied to handle zero probability problems. Smoothing ensures that unseen word combinations do not result in zero probability, making the model more robust. These models demonstrate how increasing the value of N improves contextual understanding but also increases data requirements.

Sentence Probability Results:

Sentence probabilities were calculated for multiple test sentences using Unigram, Bigram, and Trigram models. The Unigram model generally produced very small probabilities because it ignores word order and context. The Bigram model provided better probability estimates as it considers relationships between consecutive words. The Trigram model gave the most accurate probabilities since it uses more contextual information.

However, in some cases, Trigram probabilities were extremely low due to insufficient training data. This highlights a key limitation of higher-order N-gram models: they require large datasets to perform well. Despite this limitation, Trigram models usually produce more meaningful results for well-represented word sequences. The experiment clearly shows how context improves prediction quality in language models.

Perplexity Comparison:

Perplexity was used as the main evaluation metric to compare the performance of different models. Perplexity measures how well a language model predicts a given sentence. A lower perplexity value indicates that the model is less confused and performs better. In this experiment, the Trigram model generally produced the lowest perplexity values, indicating superior performance. The Bigram model showed moderate perplexity, while the Unigram model had the highest perplexity.

This result confirms that models using more context perform better in predicting word sequences. However, the Trigram model’s performance depends heavily on the availability of sufficient data. In small datasets, Trigram models may suffer from data sparsity, making smoothing essential. Overall, perplexity proved to be an effective measure for comparing language models.

Observations and Conclusion:

From this experiment, it can be concluded that N-gram models are simple yet powerful tools for language modeling. The Unigram model is easy to implement but lacks contextual understanding, making it unsuitable for real-world applications. The Bigram model offers a good balance between simplicity and performance and is widely used in practical systems. The Trigram model performs best when sufficient data is available but requires more computational resources.

Smoothing plays a vital role in improving model performance by handling unseen words and preventing zero probability issues. The experiment also highlights the importance of dataset size in training accurate language models. In real-world applications, large corpora and advanced smoothing techniques are used to overcome the limitations of simple N-gram models. Overall, this experiment provides a strong foundation for understanding more advanced language modeling techniques such as neural language models and deep learning-based approaches.