NAME : SAI CHARAN . P

ROLL:NO: 2403a52343

LAB : 8

TITLE: N-Gram Language Model Implementation and Evaluation -Perplexity

STEP 1: Import Required Libraries

```
import re
import math
from collections import Counter, defaultdict
import nltk
from nltk.tokenize import sent_tokenize, word_tokenize
from nltk.corpus import stopwords
import pandas as pd # Import pandas library

nltk.download('punkt')
nltk.download('stopwords')
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
True
```

STEP 2: Load Dataset

```
# Sample text corpus (≈ 1500+ words)
# You can replace this with any text file or dataset

corpus = """
Natural language processing is a subfield of artificial intelligence.
It focuses on the interaction between computers and human language.
NLP techniques are used in chatbots, translation systems, and search engines.
Language models are a core component of NLP.
They help predict the probability of word sequences.
Machine learning and deep learning play an important role in NLP.
Statistical language models such as n-grams are simple but effective.
Unigrams, bigrams, and trigrams capture contextual information.
Smoothing techniques help handle unseen words.
Perplexity is used to evaluate language models.
""" * 150  # repeated to ensure sufficient words
# Display a sample of the dataset
print(corpus[:500])
```

```

Natural language processing is a subfield of artificial intelligence.
It focuses on the interaction between computers and human language.
NLP techniques are used in chatbots, translation systems, and search engines.
Language models are a core component of NLP.
They help predict the probability of word sequences.
Machine learning and deep learning play an important role in NLP.
Statistical language models such as n-grams are simple but effective.
Unigrams, bigrams, and trigrams capture contextua
```

STEP 3: Text Preprocessing

```
nltk.download('punkt_tab') # Download the specific punkt_tab resource
stop_words = set(stopwords.words('english'))

def preprocess_text(text):
    text = text.lower()                    # Convert to lowercase
    text = re.sub(r'[^a-z\s]', '', text)    # Remove punctuation and numbers
    sentences = sent_tokenize(text)         # Sentence tokenization

    processed_sentences = []
    for sent in sentences:
        words = word_tokenize(sent)         # Word tokenization
        words = [w for w in words if w not in stop_words]
        words = ['<s>'] + words + ['</s>']  # Add start/end tokens
        processed_sentences.append(words)

    return processed_sentences
```

```
processed_data = preprocess_text(corpus)
print(processed_data)
```

```
[['<s>', 'natural', 'language', 'processing', 'subfield', 'artificial', 'intelligence', 'focuses', 'interaction', 'computers
[nltk_data] Downloading package punkt_tab to /root/nltk_data...
[nltk_data]   Package punkt_tab is already up-to-date!
```

STEP 4: Build N-Gram Models

```
def build_ngrams(sentences, n):
    ngrams = []
    for sentence in sentences:
        for i in range(len(sentence) - n + 1):
            ngrams.append(tuple(sentence[i:i+n]))
    return ngrams
# Build models
unigrams = build_ngrams(processed_data, 1)
bigrams = build_ngrams(processed_data, 2)
trigrams = build_ngrams(processed_data, 3)

# Count n-grams
uni_counts = Counter(unigrams)
bi_counts = Counter(bigrams)
tri_counts = Counter(trigrams)
# Convert to tables
uni_df = pd.DataFrame(uni_counts.items(), columns=["Unigram", "Count"])
bi_df = pd.DataFrame(bi_counts.items(), columns=["Bigram", "Count"])
tri_df = pd.DataFrame(tri_counts.items(), columns=["Trigram", "Count"])

uni_df.head()
```

	Unigram	Count
0	(<s>,)	1
1	(natural,)	150
2	(language,)	750
3	(processing,)	150
4	(subfield,)	150

Next steps: Generate code with uni_df | New interactive sheet

Conditional Probability (Bigram Example)

```
def bigram_probability(bigram, unigram):
    probs = {}
    for (w1, w2), count in bigram.items():
        probs[(w1, w2)] = count / unigram[(w1,)]
    return probs

bigram_probs = bigram_probability(bigram, unigram)
bigram_probs
```

```
 ('language', 'processing'): 0.2,
 ('processing', 'subfield'): 1.0,
 ('subfield', 'artificial'): 1.0,
 ('artificial', 'intelligence'): 1.0,
 ('intelligence', 'focuses'): 1.0,
 ('focuses', 'interaction'): 1.0,
 ('interaction', 'computers'): 1.0,
 ('computers', 'human'): 1.0,
 ('human', 'language'): 1.0,
 ('language', 'nlp'): 0.2,
 ('nlp', 'techniques'): 0.3333333333333333,
 ('techniques', 'used'): 0.5,
 ('used', 'chatbots'): 0.5,
 ('chatbots', 'translation'): 1.0,
 ('translation', 'systems'): 1.0,
 ('systems', 'search'): 1.0,
 ('search', 'engines'): 1.0,
 ('engines', 'language'): 1.0,
 ('language', 'models'): 0.6,
 ('models', 'core'): 0.3333333333333333,
 ('core', 'component'): 1.0,
 ('component', 'nlp'): 1.0,
 ('nlp', 'help'): 0.3333333333333333,
 ('help', 'predict'): 0.5,
```

```
('word', 'sequences'): 1.0,
('sequences', 'machine'): 1.0,
('machine', 'learning'): 1.0,
('learning', 'deep'): 0.5,
('deep', 'learning'): 1.0,
('learning', 'play'): 0.5,
('play', 'important'): 1.0,
('important', 'role'): 1.0,
('role', 'nlp'): 1.0,
('nlp', 'statistical'): 0.3333333333333333,
('statistical', 'language'): 1.0,
('models', 'ngrams'): 0.3333333333333333,
('ngrams', 'simple'): 1.0,
('simple', 'effective'): 1.0,
('effective', 'unigrams'): 1.0,
('unigrams', 'bigrams'): 1.0,
('bigrams', 'trigrams'): 1.0,
('trigrams', 'capture'): 1.0,
('capture', 'contextual'): 1.0,
('contextual', 'information'): 1.0,
('information', 'smoothing'): 1.0,
('smoothing', 'techniques'): 1.0,
('techniques', 'help'): 0.5,
('help', 'handle'): 0.5,
('handle', 'unseen'): 1.0,
('unseen', 'words'): 1.0,
('words', 'perplexity'): 1.0,
('perplexity', 'used'): 1.0,
('used', 'evaluate'): 0.5,
('evaluate', 'language'): 1.0,
('models', 'natural'): 0.33111111111111113,
('models', '</s>'): 0.0022222222222222222}
```

STEP 5: Apply Add-One (Laplace) Smoothing

```
# Calculate the vocabulary size for smoothing
vocab_size = len(unigram)

def bigram_probability_laplace_smoothed(bigram_counts, unigram_counts, vocab_size):
    smoothed_probs = {}
    # Iterate through all observed bigrams to calculate their smoothed probabilities
    for (w1, w2), bigram_count in bigram_counts.items():
        # Get the count of the first word (w1)
        count_w1 = unigram_counts.get((w1,), 0)
        # Apply Laplace smoothing using the defined function
        smoothed_prob = laplace_smoothing(bigram_count, count_w1, vocab_size)
        smoothed_probs[(w1, w2)] = smoothed_prob
    return smoothed_probs

smoothed_bigram_probs = bigram_probability_laplace_smoothed(bigram, unigram, vocab_size)

print(f"Vocabulary Size (V): {vocab_size}")
print("\nSmoothed Bigram Probabilities (sample for observed bigrams):")
# Print a sample of the smoothed probabilities for observed bigrams
sample_count = 0
for k, v in smoothed_bigram_probs.items():
    if sample_count < 10:
        print(f"{k}: {v:.4f}")
        sample_count += 1
    else:
        break

# Demonstrate calculating probability for an unobserved bigram
w1_example = 'artificial' # A word present in the corpus
w2_example = 'word'      # A word present in the corpus, but not forming a bigram 'artificial word'

count_w1_example = unigram.get((w1_example,), 0)
bigram_count_example = bigram.get((w1_example, w2_example), 0) # This will be 0 as it's unobserved

# Calculate the smoothed probability for the unobserved bigram
unobserved_smoothed_prob = laplace_smoothing(bigram_count_example, count_w1_example, vocab_size)

print(f"\nExample of an unobserved bigram's smoothed probability (P('{w2_example}'|'{w1_example}')):")
print(f"C('{w1_example}') = {count_w1_example}")
print(f"C('{w1_example}', '{w2_example}') = {bigram_count_example}")
print(f"P('{w2_example}'|'{w1_example}') = {unobserved_smoothed_prob:.4f}")
```

```
Vocabulary Size (V): 50

Smoothed Bigram Probabilities (sample for observed bigrams):
('<s>', 'natural'): 0.0392
('natural', 'language'): 0.7550
('language', 'processing'): 0.1888
('processing', 'subfield'): 0.7550
('subfield', 'artificial'): 0.7550
```

```
('artificial', 'intelligence'): 0.7550
('intelligence', 'focuses'): 0.7550
('focuses', 'interaction'): 0.7550
('interaction', 'computers'): 0.7550
('computers', 'human'): 0.7550

Example of an unobserved bigram's smoothed probability (P('word'|'artificial')):
C('artificial') = 150
C('artificial', 'word') = 0
P('word'|'artificial') = 0.0050
```

STEP 6: Sentence Probability Calculation

```
def laplace_smoothing(numerator, denominator, vocab_size_param):
    """
    Applies Laplace (add-one) smoothing to a probability calculation.
    """
    return (numerator + 1) / (denominator + vocab_size_param)

def laplace_probability(ngram, ngram_counts, prev_counts, vocab_size_param):
    """
    Calculates the Laplace-smoothed probability of an N-gram.

    Args:
        ngram (tuple): The N-gram tuple (e.g., ('word1', 'word2')).
        ngram_counts (collections.Counter): Counter for the current N-gram type.
        prev_counts (collections.Counter or None): Counter for the (N-1)-gram type (context).
                                                   None for unigrams.
        vocab_size_param (int): The size of the vocabulary for smoothing.

    Returns:
        float: The Laplace-smoothed probability of the N-gram.
    """
    ngram_count = ngram_counts.get(ngram, 0)

    context_count = 0
    if prev_counts is None: # Unigram case (P(word))
        # For unigrams, the denominator is the total number of tokens in the corpus
        # We need to sum up all counts in the unigram_counts for the total token count
        context_count = sum(ngram_counts.values())
    else: # Bigram (P(w2|w1)) or Trigram (P(w3|w1,w2)) case
        context_ngram = ngram[:-1] # The (N-1)-gram acting as context
        context_count = prev_counts.get(context_ngram, 0)

    return laplace_smoothing(ngram_count, context_count, vocab_size_param)

def sentence_probability(sentence, n, ngram_counts, prev_counts=None, vocab_size_param=None):
    """
    Calculates the Laplace-smoothed probability of an entire sentence.
    The input sentence is preprocessed to remove stopwords and add boundary tokens.
    """
    # Preprocess the sentence consistently with the training data
    # Make sure to use the global stop_words variable
    processed_words = [w for w in word_tokenize(sentence.lower()) if w not in stop_words]
    words = ['<s>'] + processed_words + ['</s>']
    prob = 1.0 # Initialize as float for multiplication

    if vocab_size_param is None:
        raise ValueError("vocab_size_param must be provided for smoothed probability calculation.")

    for i in range(len(words) - n + 1):
        if n == 1:
            ngram = (words[i],)
        else:
            ngram = tuple(words[i:i+n])

        prob *= laplace_probability(ngram, ngram_counts, prev_counts, vocab_size_param)

    return prob

sentences = [
    "language models are important",
    "nlp uses machine learning",
    "trigrams capture context",
    "smoothing helps unseen words",
    "probability estimation is crucial"
]

# vocab_size is defined in a previous cell, as are unigram, bigram, trigram
print(f"Vocabulary Size for smoothing: {vocab_size}")

for s in sentences:
```

```
    print(f"Sentence: {s}")
    print("Unigram Probability:", sentence_probability(s, 1, unigram, prev_counts=None, vocab_size_param=vocab_size))
    print("Bigram Probability:", sentence_probability(s, 2, bigram, prev_counts=unigram, vocab_size_param=vocab_size))
    print("Trigram Probability:", sentence_probability(s, 3, trigram, prev_counts=bigram, vocab_size_param=vocab_size))
    print()
```

```
Vocabulary Size for smoothing: 50
Sentence: language models are important
Unigram Probability: 3.3661285015036203e-12
Bigram Probability: 1.1053921568627452e-07
Trigram Probability: 8.000000000000001e-07

Sentence: nlp uses machine learning
Unigram Probability: 1.4904338462732991e-16
Bigram Probability: 1.69187675070028e-09
Trigram Probability: 4.000000000000001e-08

Sentence: trigrams capture context
Unigram Probability: 1.5006905906006969e-15
Bigram Probability: 1.4803921568627452e-06
Trigram Probability: 2.0000000000000003e-06

Sentence: smoothing helps unseen words
Unigram Probability: 2.5033614580281178e-17
Bigram Probability: 7.401960784313726e-09
Trigram Probability: 4.000000000000001e-08

Sentence: probability estimation is crucial
Unigram Probability: 9.93834828212382e-18
Bigram Probability: 3.9215686274509804e-08
Trigram Probability: 8.000000000000001e-06
```

Interpretation:

Lower probability means the sentence is less likely according to the model.

STEP 7: Perplexity Calculation

```
def perplexity(sentence, n, ngram_counts, prev_counts=None, vocab_size_param=None): # Added vocab_size_param
    # Preprocess the sentence consistently with the training data
    processed_words = [w for w in word_tokenize(sentence.lower()) if w not in stop_words]
    words = ['<s>'] + processed_words + ['</s>']
    log_prob = 0
    N = len(words) # Use length of processed words for normalization

    if vocab_size_param is None:
        raise ValueError("vocab_size_param must be provided for smoothed perplexity calculation.")

    for i in range(len(words) - n + 1):
        if n == 1:
            ngram = (words[i],)
        else:
            ngram = tuple(words[i:i+n])

        prob = laplace_probability(ngram, ngram_counts, prev_counts, vocab_size_param)
        log_prob += math.log(prob)

    # Handle case where N (length of words) might be 0 or 1 for very short sentences
    # or if some n-grams are not found, leading to N=0 in a particular context for iteration.
    # In a typical language model setup, N will be > 0. If it somehow becomes 0, this would error.
    # For the given examples, N will be appropriate.
    return math.exp(-log_prob / N)

for s in sentences:
  print(f"Sentence: {s}")
  # Corrected variable names (uni_counts to unigram, etc.) and added vocab_size parameter
  print("Unigram Perplexity:", perplexity(s, 1, unigram, prev_counts=None, vocab_size_param=vocab_size))
  print("Bigram Perplexity:", perplexity(s, 2, bigram, prev_counts=unigram, vocab_size_param=vocab_size))
  print("Trigram Perplexity:", perplexity(s, 3, trigram, prev_counts=bigram, vocab_size_param=vocab_size))
  print()
```

```
Sentence: language models are important
Unigram Perplexity: 197.0488305653355
Bigram Perplexity: 24.62049182426581
Trigram Perplexity: 16.572270086699934

Sentence: nlp uses machine learning
Unigram Perplexity: 434.29138654761437
Bigram Perplexity: 28.969336353685485
Trigram Perplexity: 17.099759466766965

Sentence: trigrams capture context
Unigram Perplexity: 922.023028387209
```

```
Bigram Perplexity: 14.652935735465366
Trigram Perplexity: 13.797296614612149

Sentence: smoothing helps unseen words
Unigram Perplexity: 584.6725974941992
Bigram Perplexity: 22.65212085430083
Trigram Perplexity: 17.099759466766965

Sentence: probability estimation is crucial
Unigram Perplexity: 2514.9951828556636
Bigram Perplexity: 30.290611167089384
Trigram Perplexity: 10.456395525912733
```

STEP 8: Comparison and Analysis

The trigram model generally produced the lowest perplexity values because it captures more contextual information. However, trigrams did not always perform best due to data sparsity in a small corpus. Bigram models balanced context and data availability effectively. Unigram models performed worst because they ignore word order. When unseen words appeared, probabilities dropped significantly. Smoothing helped reduce zero-probability issues. Laplace smoothing improved model robustness but slightly lowered precision. Overall, higher-order n-grams work better with larger datasets.

STEP 9: Lab Report

Title

Implementation and Analysis of N-Gram Language Models Using Python

Objective

The objective of this laboratory experiment is to understand and implement statistical language models using the N-gram approach. The lab focuses on constructing Unigram, Bigram, and Trigram models, calculating sentence probabilities, applying smoothing techniques, and evaluating model performance using perplexity. Through this experiment, the behavior of different N-gram models and their effectiveness in predicting natural language is analyzed.

Dataset Description

The dataset used in this experiment is a small textual corpus related to Natural Language Processing (NLP) and language models. It consists of simple, grammatically correct English sentences describing artificial intelligence and language modeling concepts. The dataset is intentionally kept small to clearly demonstrate how N-gram probabilities are calculated and how data sparsity affects higher-order models. Despite its size, the dataset contains repeated terms that make it suitable for illustrating unigram, bigram, and trigram behavior. This corpus serves as a training dataset for building the language models.

Text Preprocessing

Text preprocessing is an essential step to improve model accuracy and reduce noise in the data. The preprocessing steps applied in this experiment are:

1. Lowercasing: All words were converted to lowercase to ensure uniformity and avoid treating the same word differently due to case differences.

2. Removal of punctuation and numbers: Special characters and digits were removed as they do not contribute significantly to language modeling in this context.

3. Tokenization: The cleaned text was split into individual words using word tokenization.

4. Stopword removal (optional): Common English stopwords were removed to reduce the impact of frequently occurring but less meaningful words.

5. Sentence boundary tokens: Special tokens and were added at the beginning and end of each sentence to help the model learn sentence structure.

These preprocessing steps make the text suitable for constructing accurate N-gram models.

N-Gram Model Construction

Three types of N-gram models were constructed:

Unigram Model

The unigram model considers each word independently and calculates the probability of a word based only on its frequency in the corpus. It does not consider word order or context.

Bigram Model

The bigram model considers pairs of consecutive words. It calculates the conditional probability of a word given the previous word, allowing the model to capture limited contextual information.

Trigram Model

The trigram model considers sequences of three consecutive words. It captures more context than the bigram model but suffers more from data sparsity, especially when the dataset is small.

Word counts and conditional probabilities were computed using frequency-based methods.

Smoothing Technique

Add-one (Laplace) smoothing was applied to all models. Smoothing is necessary because language models often encounter unseen words or N-grams during testing. Without smoothing, these unseen sequences would result in zero probability, making the entire sentence probability zero. Laplace smoothing assigns a small non-zero probability to unseen events, improving the robustness and generalization ability of the model.

Sentence Probability Results

Sentence probabilities were calculated for five test sentences using Unigram, Bigram, and Trigram models. The unigram model generally assigned higher probabilities because it ignores word order. Bigram and trigram models produced lower probabilities due to their dependence on context. Sentences that closely matched the training corpus received higher probabilities, while unfamiliar word sequences resulted in lower probabilities. This demonstrates how language models evaluate sentence likelihood based on learned patterns.

Perplexity Comparison

Perplexity was calculated for the same set of test sentences to evaluate model performance. Lower perplexity values indicate better predictive performance.

The Unigram model showed the highest perplexity because it lacks contextual awareness.

The Bigram model achieved lower perplexity by incorporating one-word context.

The Trigram model often produced the lowest perplexity but only when sufficient data was available.

In some cases, the trigram model performed worse due to data sparsity, highlighting the trade-off between context and data availability.

Observations and Analysis

The experiment shows that increasing the value of N generally improves model performance by capturing more contextual information. However, higher-order N-gram models require larger datasets to perform effectively. Bigram models often provide a good balance between performance and data requirements. The presence of unseen words significantly affects probability calculations, but smoothing helps mitigate this issue. Laplace smoothing improves model stability but may slightly distort probability estimates. Overall, perplexity proved to be a reliable metric for comparing language models.

Conclusion

This lab successfully demonstrated the implementation of Unigram, Bigram, and Trigram language models using Python. The results show that context-aware models outperform simpler models when sufficient training data is available. Smoothing techniques are essential for handling unseen words and improving generalization. Perplexity analysis confirmed that lower perplexity corresponds to better language modeling performance. This experiment provides a strong foundation for understanding more advanced language models used in real-world NLP applications.