**University of Central Missouri**

**Department of Computer Science & Cybersecurity**

**CS5760 Natural Language Processing**

**Fall 2025**

**Homework 2.**

**Student name: Neha Reddy Palem**

**Submission Requirements:**

* Once finished your assignment push your source code to your repo (GitHub) and explain the work through the ReadMe file properly. Make sure you add your student info in the ReadMe file.
* Submit your GitHub link on the Bright Space.
* Comment your code appropriately ***IMPORTANT.***
* Any submission after provided deadline is considered as a late submission.

**Q1. Worked Example Document Classification (based on slide: Test document “predictable no fun”)**  
Using the smoothed likelihoods and priors from Q2, compute the probability scores for the document *“predictable no fun”* under both the positive and negative classes.

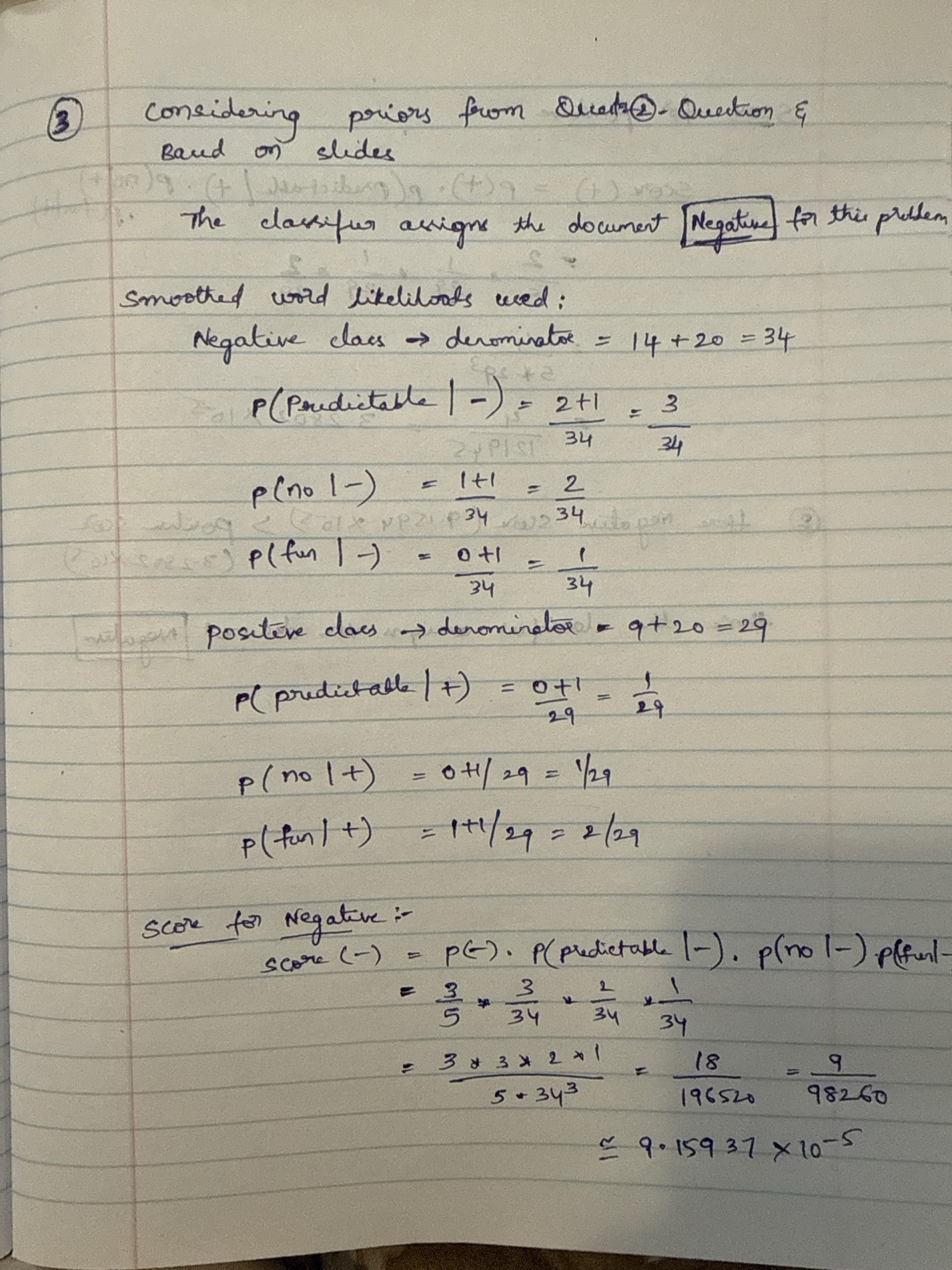
### **Tasks:**

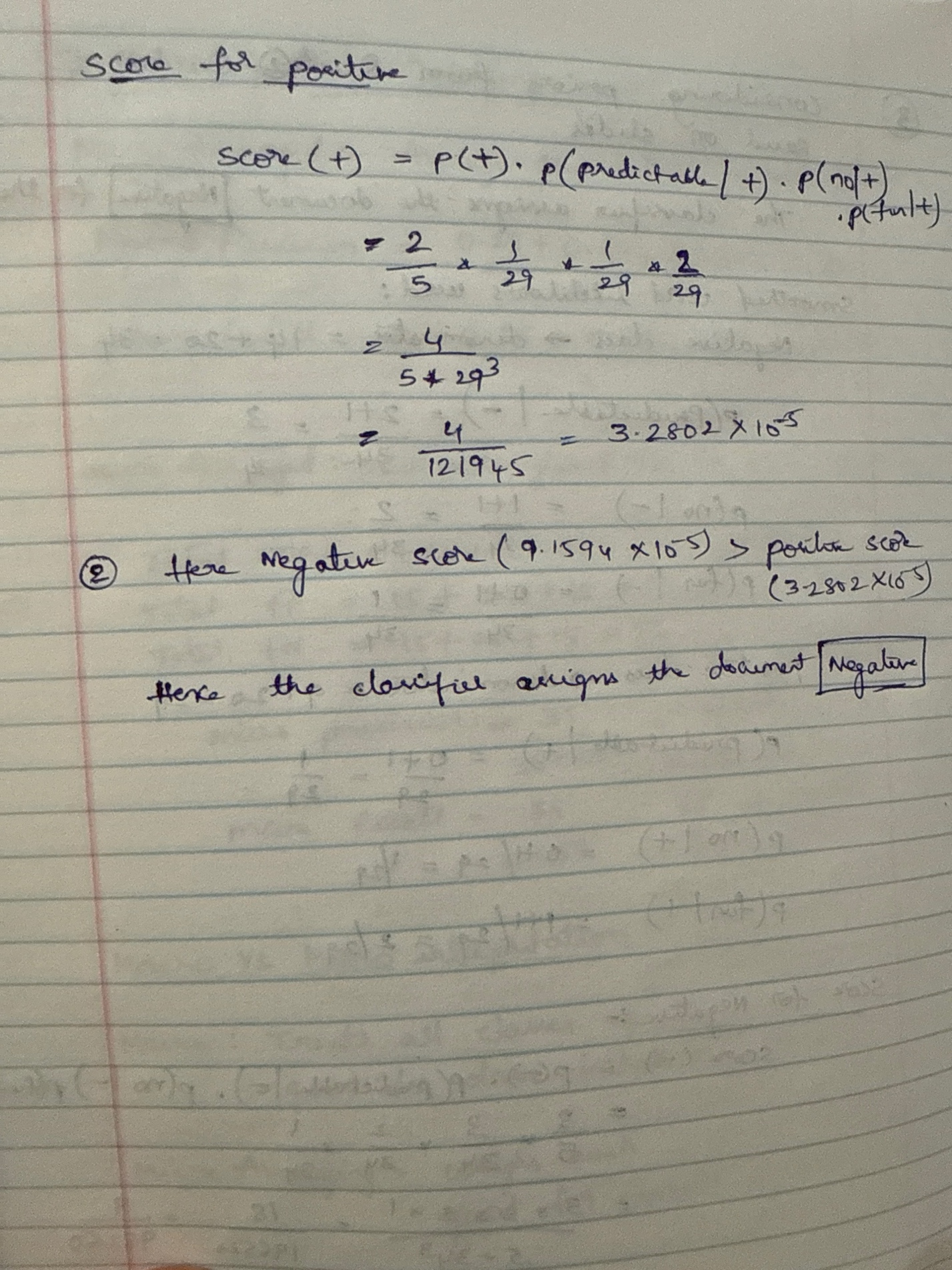
1. Show each step of the multiplication.
2. Which class should the system assign to this document

**Ans:**

The classifier assigns the document **Negative** for this problem

\*\*\*Ans in next page\*\*\*

****



**Q2. Harms of Classification (based on slide: Avoiding Harms in Classification)**

### **Tasks:**

1. Define **representational harm** and explain how the Kiritchenko & Mohammad (2018) study demonstrates this type of harm.
2. What is one risk of censorship in toxicity classification systems (based on Dixon et al. 2018, Oliva et al. 2021)?
3. Give one reason why classifiers may perform worse on African American English or Indian English, even though they are varieties of English.

Ans:

1. **Representational harm**
   1. Representational harm occurs when a classifier reinforces stereotypes or misrepresents groups.
   2. Kiritchenko & Mohammad (2018) showed that tweets mentioning groups such as women, Muslims, or African Americans were more often rated as negative, even when neutral — an example of representational harm.
2. **Risk of censorship in toxicity classification**
   1. A key risk is that harmless speech gets censored.
   2. Dixon et al. (2018) found the word “gay” was flagged as toxic even in positive uses.
   3. Oliva et al. (2021) showed such errors can silence marginalized voices.
3. **Why classifiers may perform worse on AAE or Indian English**
   1. Training data is dominated by Standard American/British English.
   2. This causes a domain mismatch, since the model does not learn the vocabulary or grammar patterns of African American English (AAE) or Indian English.
   3. As a result, these valid varieties of English are more often misclassified.

**Q5: Evaluation Metrics from a Multi-Class Confusion Matrix**

The system classified 90 animals into Cat, Dog, or Rabbit. The results are shown below:

| **System \ Gold** | **Cat** | **Dog** | **Rabbit** |
| --- | --- | --- | --- |
| **Cat** | 5 | 10 | 5 |
| **Dog** | 15 | 20 | 10 |
| **Rabbit** | 0 | 15 | 10 |

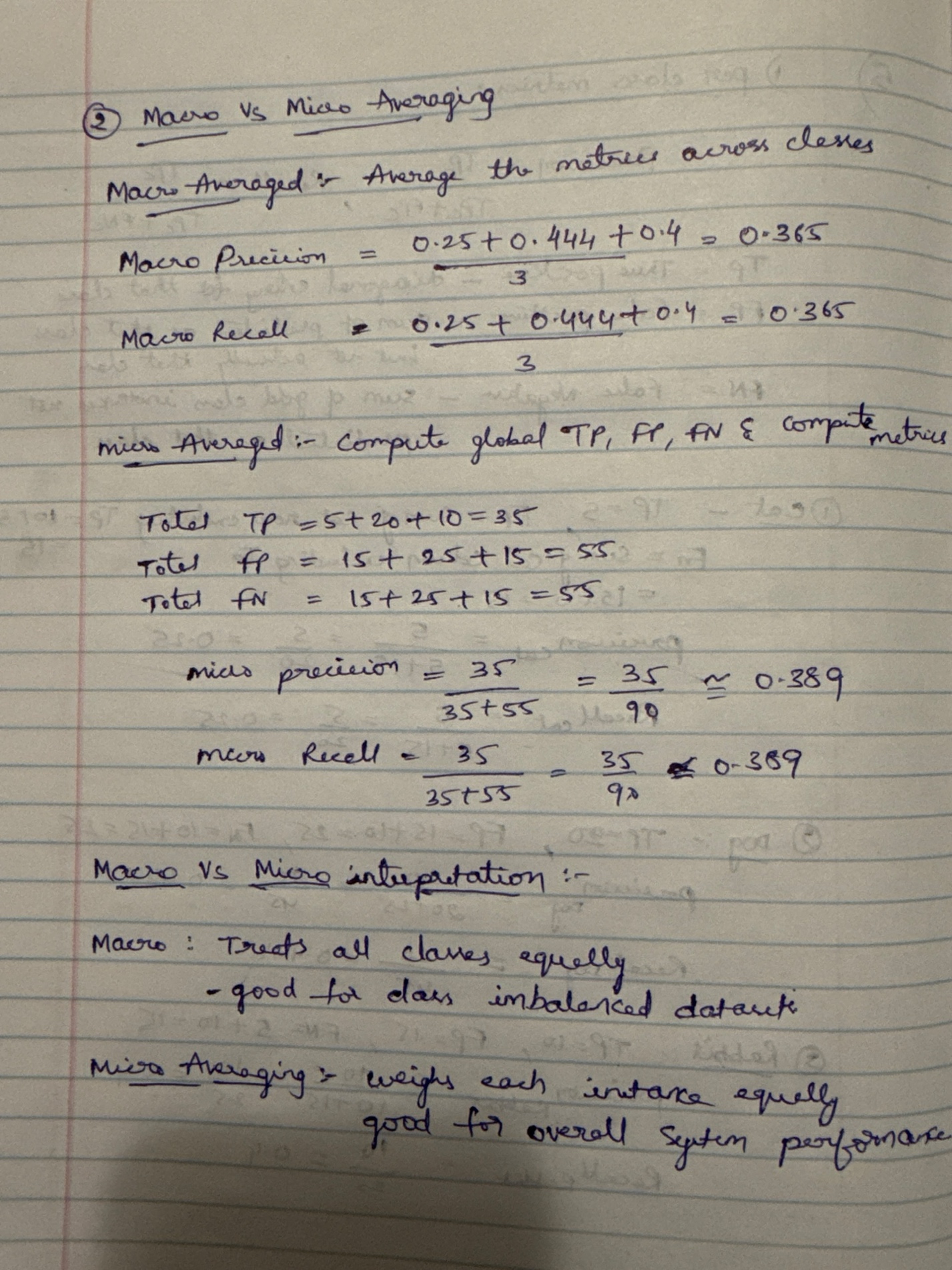
Tasks:

1. Per-Class Metrics
   * Compute precision and recall for each class (Cat, Dog, Rabbit).
2. Macro vs. Micro Averaging
   * Compute the macro-averaged precision and recall.
   * Compute the micro-averaged precision and recall.
   * Briefly explain the difference in interpretation between macro and micro averaging.
3. Programming Implementation  
   Write Python code that:
   * Accepts the confusion matrix above as input.
   * Computes per-class precision and recall.
   * Computes macro-averaged and micro-averaged precision and recall.
   * Prints all results clearly.

Answers in next page

A notebook with writing on it

AI-generated content may be incorrect.



3.Programming Implementation:

import numpy as np

# -----------------------------------------------------------

# Confusion matrix setup

# Rows = system predictions, Columns = actual gold labels

# Example: matrix[0,1] = number of items predicted as "Cat"

# but actually "Dog"

# -----------------------------------------------------------

conf\_matrix = np.array([

[5, 10, 5], # Predicted as Cat

[15, 20, 10], # Predicted as Dog

[0, 15, 10] # Predicted as Rabbit

])

# Number of classes and class labels

num\_classes = conf\_matrix.shape[0]

classes = ['Cat', 'Dog', 'Rabbit']

# -----------------------------------------------------------

# Per-class precision and recall

# Precision = TP / (TP + FP)

# Recall = TP / (TP + FN)

# -----------------------------------------------------------

precision = []

recall = []

for i in range(num\_classes):

# True Positives = correct predictions for this class

TP = conf\_matrix[i, i]

# False Positives = predicted as this class but actually other classes

FP = conf\_matrix[i, :].sum() - TP

# False Negatives = actually this class but predicted as others

FN = conf\_matrix[:, i].sum() - TP

# Compute precision and recall safely (avoid division by zero)

p = TP / (TP + FP) if (TP + FP) != 0 else 0

r = TP / (TP + FN) if (TP + FN) != 0 else 0

precision.append(p)

recall.append(r)

print(f"{classes[i]} -> Precision: {p:.3f}, Recall: {r:.3f}")

# -----------------------------------------------------------

# Macro-averaged metrics

# (Treat each class equally by averaging their metrics)

# -----------------------------------------------------------

macro\_precision = np.mean(precision)

macro\_recall = np.mean(recall)

print(f"\nMacro-averaged Precision: {macro\_precision:.3f}")

print(f"Macro-averaged Recall: {macro\_recall:.3f}")

# -----------------------------------------------------------

# Micro-averaged metrics

# (Aggregate counts across all classes first, then compute metrics)

# -----------------------------------------------------------

# Total True Positives = sum of diagonal elements

TP\_total = np.trace(conf\_matrix)

# Total False Positives = all predictions - correct ones

FP\_total = conf\_matrix.sum(axis=1).sum() - TP\_total

# Total False Negatives = all actual labels - correct ones

FN\_total = conf\_matrix.sum(axis=0).sum() - TP\_total

# Compute micro precision and recall

micro\_precision = TP\_total / (TP\_total + FP\_total)

micro\_recall = TP\_total / (TP\_total + FN\_total)

print(f"\nMicro-averaged Precision: {micro\_precision:.3f}")

print(f"Micro-averaged Recall: {micro\_recall:.3f}")

# **Q3. Bigram Probabilities and the Zero-Probability Problem**

You are given the following bigram counts from a small training corpus:

| **Previous word** | **Next words (with counts)** |
| --- | --- |
| <s> | I: 2 , deep: 1 |
| I | love: 2 |
| love | NLP: 1 , deep: 1 |
| deep | learning: 2 |
| learning | </s>: 1 , is: 1 |
| NLP | </s>: 1 |
| is | fun: 1 |
| fun | </s>: 1 |
| ate | lunch: 6 , dinner: 3 , a: 2 , the: 1 |

### **Tasks:**

1. **Bigram Sentence Probabilities**  
   Using maximum likelihood estimation (MLE):

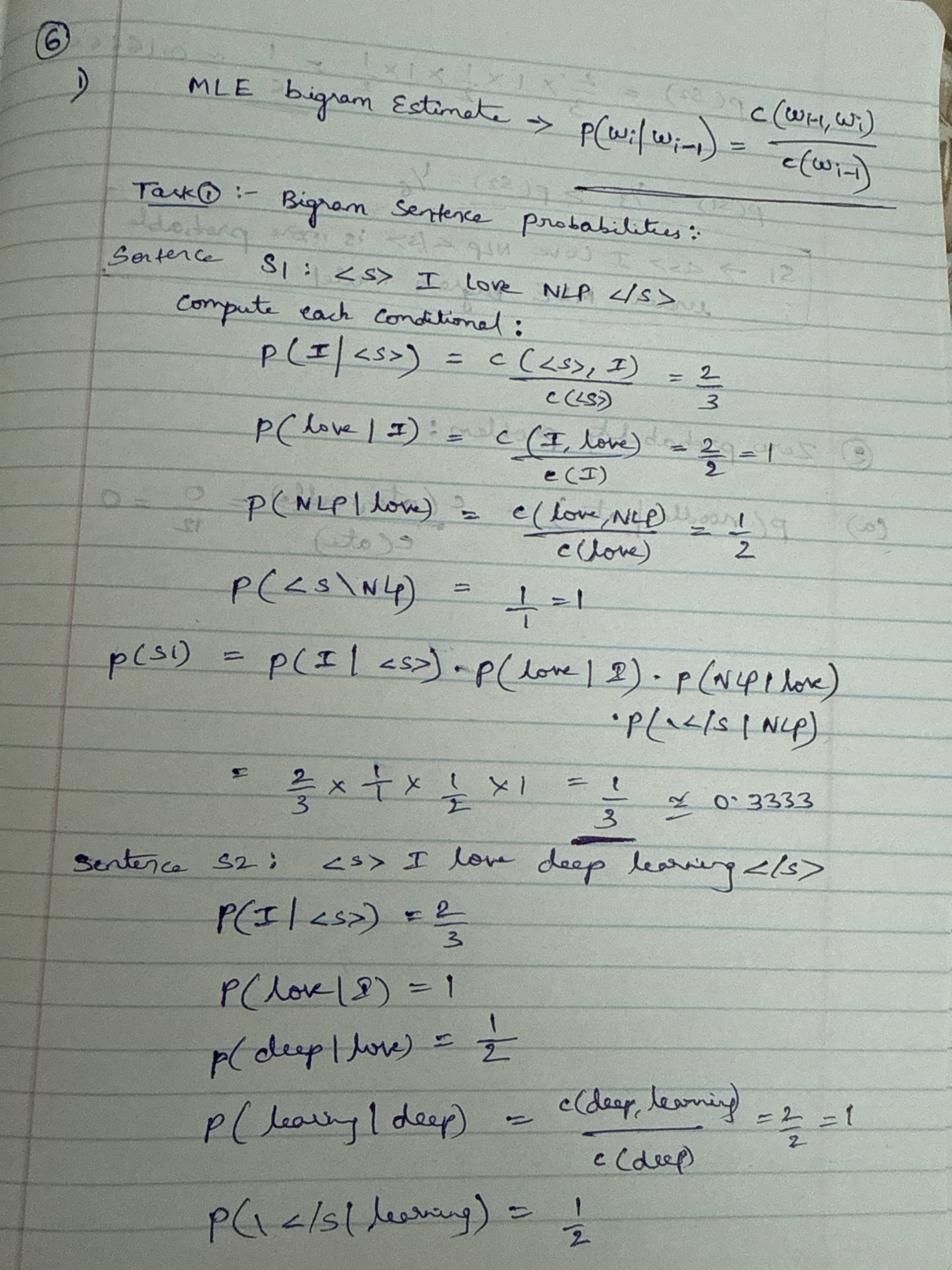
A black and white math equation

AI-generated content may be incorrect.

* + Compute the probability of sentence **S1:** <s> I love NLP </s>.
  + Compute the probability of sentence **S2:** <s> I love deep learning </s>.
  + Which sentence is more probable under the bigram model?

1. **Zero-Probability Problem**  
   Using the same table, compute:
   * P(noodle∣ate) with MLE.
   * Explain why this probability creates problems when computing sentence probabilities or perplexity.
   * Apply **Laplace smoothing (Add-1)** to recompute P(noodle∣ate). Assume the vocabulary size is 10 and total count after “ate” is 12.

**ANS: in next page**

****

### **A notebook with writing on it AI-generated content may be incorrect.**

**(b) Why is this a problem?**

* If **any** bigram in a sentence has MLE probability 0, the **product** for the whole sentence probability becomes 0. That means the model assigns zero probability to any sentence containing an unseen bigram, even if the rest of the sentence is plausible.
* In log-space this shows up as (log(0)), so you cannot compute a finite log-probability or cross-entropy.
* For evaluation metrics like **perplexity**, a zero sentence-probability makes perplexity undefined or infinite — making model comparison and evaluation fail on realistic test data where unseen bigrams are common.
* In short: MLE **does not** generalize to unseen events — leads to brittle behavior.

**(c) Apply Laplace (add-1) smoothing:**

Given: vocabulary size V=10, and total count after “ate” = 12 -> c(ate) = 12

Add-1 smoothing:

P(noodle | ate) = c(ate, noodle)+1 / c(ate) + v = 0+1/12+10 = 1 / 22 = 0.045

So with add-1 smoothing the previously zero probability becomes **≈ 0.04545,** preventing zeroing out sentence probabilities.

### **Q4. Backoff Model (based on “Activity: <s> I like cats … You like dogs” slide)**

Training corpus:

<s> I like cats </s>

<s> I like dogs </s>

<s> You like cats </s>

Counts:

* I like = 2
* You like = 1
* like cats = 2
* like dogs = 1
* cats </s> = 2
* dogs </s> = 1

### **Tasks:**

1. Compute P(cats∣I,like).
2. Compute P(dogs∣You,like) using trigram → bigram backoff.
3. Explain why backoff is necessary in this example.

**ANS:**

Given counts

* “I like” = 2
* “You like” = 1
* “like cats” = 2
* “like dogs” = 1
* “cats </s>” = 2
* “dogs </s>” = 1

**1.** Compute P(cats∣I,like).

This is a **trigram probability:**

From the corpus:

* “I like cats” appears once. → .
* “I like” appears twice. → .

1. Compute P(dogs∣You,like) using trigram → bigram backoff.

We need to compute the trigram probability . In a backoff model, we first try to use the trigram probability. If the trigram count is zero or unreliable (e.g., insufficient data), we "back off" to the bigram probability, possibly with a discounting factor. The backoff model can be expressed as:

If , use:

If , back off to the bigram probability:

where is a backoff weight to ensure probabilities sum to 1, but for simplicity in this problem, we’ll compute the backoff probability directly if needed, assuming unless specified (common in simple backoff models without discounting).

Trigram count: .

From the corpus, the trigram <s> You like dogs does **not** appear (the sentences are <s> You like cats </s>, <s> I like dogs </s>, and <s> I like cats </s>).

Thus, .

Bigram count: (given directly as "You like = 1", from <s> You like cats </s>).

Since the trigram count is 0, the trigram probability would be:

This is unreliable (and unrealistic for a language model, as it assigns zero probability to a plausible word). Therefore, we back off to the bigram probability .

Compute the bigram probability :

Numerator: (given as "like dogs = 1", from <s> I like dogs </s>).

Denominator: .

The word "like" appears in all three sentences: <s> I like cats </s>, <s> I like dogs </s>, <s> You like cats </s>.

In each sentence, "like" appears once, so:

Since , we use the backoff approximation:

1. Explain why backoff is necessary in this example.

Backoff is necessary because the trigram You like dogsdoes not appear in the training corpus, resulting in a trigram count of 0. This leads to a trigram probability of P(dogs | You, like)= 0 / 1= 0 which is problematic for several reasons:

1. **Sparsity of Data**: The training corpus is small (only three sentences), so many trigrams, like , are unseen. In language modelling, especially with limited data, it’s common for certain word sequences to have zero counts, even if they are plausible in the language.
2. **Zero Probability Issue**: Assigning a probability of 0 to , given implies that “dogs” is impossible in this context, which is unreasonable since “dogs” appears in similar contexts (e.g., after “I like”). This can lead to poor model performance, especially in tasks like text generation or prediction, where unseen but plausible sequences should have non-zero probabilities.
3. **Generalization**: Backoff allows the model to generalize by falling back to a lower-order model (bigrams in this case) when trigram counts are zero or unreliable. The bigram leverages the fact that “dogs” follows “like” in another sentence (<s> I like dogs </s>), providing a more robust estimate based on partial context.
4. **Smoothing Alternative**: While smoothing techniques like add-1 (Laplace) could assign a small non-zero probability to unseen trigrams, backoff is often preferred in n-gram models because it explicitly uses more reliable lower-order statistics (bigrams or unigrams) rather than uniformly distributing probability across all vocabulary words, which can dilute meaningful patterns in small datasets.

In this example, backing off to P(dogs | like)= 1 / 3, provides a reasonable probability estimate based on the observation that “dogs” follows “like” once out of three times, making the model more robust despite the absence of the full trigram.

Part 2

### **Q1. Programming: Bigram Language Model Implementation (based on “Activity: I love NLP corpus” slide)**

### **Tasks:**

Write a Python program to:

1. Read the training corpus:
2. <s> I love NLP </s>
3. <s> I love deep learning </s>
4. <s> deep learning is fun </s>
5. Compute unigram and bigram counts.
6. Estimate bigram probabilities using MLE.
7. Implement a function that calculates the probability of any given sentence.
8. Test your function on both sentences:
   * <s> I love NLP </s>
   * <s> I love deep learning </s>
9. Print which sentence the model prefers and why.

**Program:**

from collections import defaultdict

# Step 1: Training corpus

corpus = [

"<s> I love NLP </s>",

"<s> I love deep learning </s>",

"<s> deep learning is fun </s>"

]

# Tokenize sentences

tokenized\_sentences = [sentence.split() for sentence in corpus]

# Step 2: Count unigrams and bigrams

unigram\_counts = defaultdict(int)

bigram\_counts = defaultdict(int)

for sentence in tokenized\_sentences:

for i, word in enumerate(sentence):

unigram\_counts[word] += 1

if i < len(sentence) - 1:

bigram = (word, sentence[i + 1])

bigram\_counts[bigram] += 1

# Step 3: Compute bigram probabilities using MLE

# P(w2 | w1) = Count(w1, w2) / Count(w1)

bigram\_probabilities = {}

for (w1, w2), count in bigram\_counts.items():

bigram\_probabilities[(w1, w2)] = count / unigram\_counts[w1]

# Step 4: Function to calculate probability of a sentence

def sentence\_probability(sentence\_tokens):

prob = 1.0

for i in range(len(sentence\_tokens) - 1):

bigram = (sentence\_tokens[i], sentence\_tokens[i + 1])

if bigram in bigram\_probabilities:

prob \*= bigram\_probabilities[bigram]

else:

prob \*= 0 # unseen bigram → probability = 0

return prob

# Step 5: Print counts and probabilities

print("=== Unigram Counts ===")

for word, count in unigram\_counts.items():

print(f"{word}: {count}")

print("\n=== Bigram Counts ===")

for bigram, count in bigram\_counts.items():

print(f"{bigram}: {count}")

print("\n=== Bigram Probabilities (MLE) ===")

for (w1, w2), prob in bigram\_probabilities.items():

print(f"P({w2}|{w1}) = {prob:.4f}")

# Step 6: Test on given sentences

test\_sent1 = "<s> I love NLP </s>".split()

test\_sent2 = "<s> I love deep learning </s>".split()

prob1 = sentence\_probability(test\_sent1)

prob2 = sentence\_probability(test\_sent2)

print("\n=== Sentence Probabilities ===")

print("Sentence 1:", " ".join(test\_sent1), "→ Probability:", prob1)

print("Sentence 2:", " ".join(test\_sent2), "→ Probability:", prob2)

# Step 7: Compare which one is preferred

if prob1 > prob2:

print("\nThe model prefers Sentence 1 because it has a higher probability.")

elif prob2 > prob1:

print("\nThe model prefers Sentence 2 because it has a higher probability.")

else:

print("\n The model finds both sentences equally likely.")