Corpus

```
D1 = "."
D2 = "I am learning programming, data structures, and machine learning during r
D3 = "I completed my intermediate education in MPC with strong interest in math
D4 = "I want to become a software engineer and build innovative technology solu
D5 = "Artificial intelligence and machine learning are transforming modern indu
D6 = "Cybersecurity protects systems from malware, phishing attacks, and cyber 
D7 = "Cloud computing provides scalable infrastructure and distributed storage 
D8 = "Data science combines statistics, programming, and domain expertise."
D9 = "Blockchain ensures transparency and security in digital financial transac
D10 = "Quantum computing uses qubits, superposition, and entanglement principle
```

Unigram Counts

```
import collections
import string
D1 = "."
D2 = "I am learning programming, data structures, and machine learning during r
D3 = "I completed my intermediate education in MPC with strong interest in math
D4 = "I want to become a software engineer and build innovative technology solu
D5 = "Artificial intelligence and machine learning are transforming modern indu
D6 = "Cybersecurity protects systems from malware, phishing attacks, and cyber 
D7 = "Cloud computing provides scalable infrastructure and distributed storage 
D8 = "Data science combines statistics, programming, and domain expertise."
D9 = "Blockchain ensures transparency and security in digital financial transac
D10 = "Quantum computing uses qubits, superposition, and entanglement principle
combined_text = f"{D1} {D2} {D3} {D4} {D5} {D6} {D7} {D8} {D9} {D10}"
combined_text = combined_text.lower()
for p in string.punctuation:
    combined_text = combined_text.replace(p, "")
words = combined_text.split()
unigram_counts = collections.Counter(words)
print("Unigram Counts:\n")
for word, count in unigram_counts.most_common():
    print(f"{word} : {count}")
V = len(unigram_counts)
print("\nVocabulary Size =", V)
```

```
innovative : 1
technology : 1
solutions : 1
artificial : 1
intelligence : 1
are : 1
transforming : 1
modern : 1
industries : 1
cybersecurity : 1
protects : 1
systems : 1
from : 1
malware : 1
phishing : 1
attacks : 1
cyber : 1
threats : 1
cloud : 1
provides : 1
scalable : 1
infrastructure : 1
distributed : 1
storage : 1
services : 1
science : 1
combines : 1
statistics : 1
domain : 1
expertise : 1
blockchain : 1
ensures : 1
transparency : 1
security : 1
digital : 1
financial : 1
transactions : 1
quantum : 1
uses : 1
qubits : 1
superposition : 1
entanglement : 1
principles : 1

Vocabulary Size = 72
```

Bi-Gram Counts

```
import collections
import string
D1 = "."
D2 = "I am learning programming, data structures, and machine learning during r
D3 = "I completed my intermediate education in MPC with strong interest in math
D4 = "I want to become a software engineer and build innovative technology solu
```

```
D5 = "Artificial intelligence and machine learning are transforming modern indu
D6 = "Cybersecurity protects systems from malware, phishing attacks, and cyber
D7 = "Cloud computing provides scalable infrastructure and distributed storage
D8 = "Data science combines statistics, programming, and domain expertise."
D9 = "Blockchain ensures transparency and security in digital financial transa
D10 = "Quantum computing uses qubits, superposition, and entanglement principle
combined_text = f"{D1} {D2} {D3} {D4} {D5} {D6} {D7} {D8} {D9} {D10}"
combined_text = combined_text.lower()
for p in string.punctuation:
    combined_text = combined_text.replace(p, "")
words = combined_text.split()
bigrams = []
for i in range(len(words) - 1):
    bigrams.append((words[i], words[i+1]))
bigram_counts = collections.Counter(bigrams)
print("\nBigram Counts:")
for bigram, count in bigram_counts.most_common():
    print(f"{bigram[0]} {bigram[1]}: {count}")
```

```
a software: 1
software engineer: 1
engineer and: 1
and build: 1
build innovative: 1
innovative technology: 1
```

```
programming and: 1
and domain: 1
domain expertise: 1
expertise blockchain: 1
blockchain ensures: 1
ensures transparency: 1
transparency and: 1
and security: 1
security in: 1
in digital: 1
digital financial: 1
financial transactions: 1
transactions quantum: 1
quantum computing: 1
computing uses: 1
uses qubits: 1
qubits superposition: 1
superposition and: 1
and entanglement: 1
entanglement principles: 1
```

Tri-Gram Counts

```
import collections
import string
D1 = "."
D2 = "I am learning programming, data structures, and machine learning during r
D3 = "I completed my intermediate education in MPC with strong interest in math
D4 = "I want to become a software engineer and build innovative technology solu
D5 = "Artificial intelligence and machine learning are transforming modern indu
D6 = "Cybersecurity protects systems from malware, phishing attacks, and cyber
D7 = "Cloud computing provides scalable infrastructure and distributed storage
D8 = "Data science combines statistics, programming, and domain expertise."
D9 = "Blockchain ensures transparency and security in digital financial transac
D10 = "Quantum computing uses qubits, superposition, and entanglement principle
combined_text = f"{D1} {D2} {D3} {D4} {D5} {D6} {D7} {D8} {D9} {D10}"
combined_text = combined_text.lower()
for p in string.punctuation:
    combined_text = combined_text.replace(p, "")
words = combined_text.split()
trigrams = []
for i in range(len(words) - 2):
    trigrams.append((words[i], words[i+1], words[i+2]))
trigram_counts = collections.Counter(trigrams)
print("\nTrigram Counts:")
for trigram, count in trigram_counts.most_common():
    print(f"{trigram[0]} {trigram[1]} {trigram[2]}: {count}")
    technology solutions artificial: 1
    solutions artificial intelligence: 1
    artificial intelligence and: 1
    intelligence and machine: 1
    machine learning are: 1
    learning are transforming: 1
    are transforming modern: 1
    transforming modern industries: 1
    modern industries cybersecurity: 1
    industries cybersecurity protects: 1
    cybersecurity protects systems: 1
    protects systems from: 1
    systems from malware: 1
    from malware phishing: 1
    malware phishing attacks: 1
    phishing attacks and: 1
    attacks and cyber: 1
    and cyber threats: 1
    cyber threats cloud: 1
    threats cloud computing: 1
    cloud computing provides: 1
    computing provides scalable: 1
    provides scalable infrastructure: 1
    scalable infrastructure and: 1
    infrastructure and distributed: 1
    and distributed storage: 1
    distributed storage services: 1
    storage services data: 1
    services data science: 1
    data science combines: 1
    science combines statistics: 1
    combines statistics programming: 1
    statistics programming and: 1
    programming and domain: 1
    and domain expertise: 1
    domain expertise blockchain: 1
    expertise blockchain ensures: 1
    blockchain ensures transparency: 1
    ensures transparency and: 1
    transparency and security: 1
    and security in: 1
    security in digital: 1
    in digital financial: 1
    digital financial transactions: 1
    financial transactions quantum: 1
    transactions quantum computing: 1
    quantum computing uses: 1
    computing uses qubits: 1
    uses qubits superposition: 1
    qubits superposition and: 1
    superposition and entanglement: 1
    and entanglement principles: 1
```

Next word prediction using Bi-Gram Counts

```
def predict_next_word_bigram(word_sequence, bigram_counts, unigram_counts):
    words = word_sequence.lower().split()
    if not words:
        return "Please provide a word sequence."
    last_word = words[-1]
    potential_next_words = {}
    for (w1, w2), count in bigram_counts.items():
        if w1 == last_word:
            potential_next_words[w2] = count
    if not potential_next_words:
        return f"No bigram found starting with '{last_word}'."
    last_word_count = unigram_counts.get(last_word, 0)
    if last_word_count == 0:
        return f"'{last_word}' not found in unigram counts."
    predicted_word = None
    max_probability = -1
    for next_word, count in potential_next_words.items():
        probability = count / last_word_count
        print("probability of ", next_word, "is ", round(probability, 2))
        if probability > max_probability:
            max_probability = probability
            predicted_word = next_word
    return predicted_word
sequences = [
    "I am",
    "machine learning",
    "data",
    "cloud computing",
    "artificial",
    "cybersecurity",
    "blockchain",
    "quantum computing",
    "software engineer",
    "nonexistent word"
]
for seq in sequences:
    next_word = predict_next_word_bigram(seq, bigram_counts, unigram_counts)
    print(f"Given sequence: '{seq}', predicted next word: '{next_word}'\n")
```

```
probability of  learning is  1.0
Given sequence: 'I am', predicted next word: 'learning'

probability of  programming is  0.33
probability of  during is  0.33
probability of  are is  0.33
Given sequence: 'machine learning', predicted next word: 'programming'

probability of  structures is  0.5
probability of  science is  0.5
Given sequence: 'data', predicted next word: 'structures'

probability of  provides is  0.5
probability of  uses is  0.5
Given sequence: 'cloud computing', predicted next word: 'provides'

probability of  intelligence is  1.0
Given sequence: 'artificial', predicted next word: 'intelligence'
```

```
probability of  protects is  1.0
Given sequence: 'cybersecurity', predicted next word: 'protects'

probability of  ensures is  1.0
Given sequence: 'blockchain', predicted next word: 'ensures'

probability of  provides is  0.5
probability of  uses is  0.5
Given sequence: 'quantum computing', predicted next word: 'provides'

probability of  and is  1.0
Given sequence: 'software engineer', predicted next word: 'and'

Given sequence: 'nonexistent word', predicted next word: 'No bigram found
```

Deployment of Bi-Gram Model

```
ip_text=input("enter text")
next_word1 = predict_next_word_bigram(ip_text, bigram_counts, unigram_counts)
print(f"Given sequence: '{ip_text}', predicted next word: '{next_word1}'")
```

```
enter textlearning
probability of  programming is  0.33
probability of  during is  0.33
probability of  are is  0.33
Given sequence: 'learning', predicted next word: 'programming'
```

Next word prediction using Tri-Gram Counts

```
def predict_next_word_trigram(input_text, trigram_counts, bigram_counts):
    words = input_text.lower().split()
    if len(words) < 2:
        return "Please enter at least two words for trigram prediction."

    w1, w2 = words[-2], words[-1]
    candidates = {}

    for (first, second, third), freq in trigram_counts.items():
        if first == w1 and second == w2:
            candidates[third] = freq

    if not candidates:
        return f"No trigram found starting with '{w1} {w2}'."

    best_word = None
    highest_prob = 0
    # bigram_counts is a Counter of tuples
    bigram_count = bigram_counts.get((w1, w2), 0)

    if bigram_count == 0:
        return f"No matching bigram '{w1} {w2}' found."

    for word, freq in candidates.items():
            probability = freq / bigram_count
            print(f"probability of '{word}' is {round(probability, 2)}")
            if probability > highest_prob:
                highest_prob = probability
                best_word = word

    return best_word

# Test with sequences from the corpus
test_sequences = [
    "and machine",
    "i am",
    "data structures",
    "cloud computing"
]

for seq in test_sequences:
    prediction = predict_next_word_trigram(seq, trigram_counts, bigram_counts)
    print(f"Given sequence: '{seq}', predicted next word: '{prediction}'\n")
```

```
probability of 'learning' is 1.0
Given sequence: 'and machine', predicted next word: 'learning'

probability of 'learning' is 1.0
Given sequence: 'i am', predicted next word: 'learning'

probability of 'and' is 1.0
Given sequence: 'data structures', predicted next word: 'and'

probability of 'provides' is 1.0
Given sequence: 'cloud computing', predicted next word: 'provides'
```

Deployment of Tri-Gram Model

```
import collections
import string

# 1. Prepare Corpus
corpus_texts = [
    ".",
    "I am learning programming, data structures, and machine learning during my
    "I completed my intermediate education in MPC with strong interest in mathe
    "I want to become a software engineer and build innovative technology solut
    "Artificial intelligence and machine learning are transforming modern indus
    "Cybersecurity protects systems from malware, phishing attacks, and cyber t
    "Cloud computing provides scalable infrastructure and distributed storage s
    "Data science combines statistics, programming, and domain expertise.",
    "Blockchain ensures transparency and security in digital financial transact
    "Quantum computing uses qubits, superposition, and entanglement principles
]

# 2. Process Text and build N-Grams
combined_text = " ".join(corpus_texts).lower()
for p in string.punctuation:
    combined_text = combined_text.replace(p, "")
words = combined_text.split()

bigram_counts = collections.Counter(zip(words, words[1:]))
trigram_counts = collections.Counter(zip(words, words[1:], words[2:]))

# 3. Prediction Function
def predict_next_word_trigram(input_text, trigrams, bigrams):
    words = input_text.lower().split()
    if len(words) < 2: return "Please enter at least two words."
    w1, w2 = words[-2], words[-1]

    candidates = {t[2]: freq for t, freq in trigrams.items() if t[0] == w1 and
    if not candidates: return f"No trigram found for '{w1} {w2}'"

    bigram_count = bigrams.get((w1, w2), 0)
    best_word = max(candidates, key=lambda k: candidates[k] / bigram_count) if
    return best_word

# 4. Deployment
user_input = input("Enter at least two words (e.g., 'i am'): ")
if user_input.strip():
    result = predict_next_word_trigram(user_input, trigram_counts, bigram_coun
    print(f"\nInput: '{user_input}'\nPredicted next word: {result}")
```

```
Enter at least two words (e.g., 'i am'): i am

Input: 'i am'
Predicted next word: learning
```

Next Word Prediction Using Bi-Gram Counts with Laplace Smoothening

```
    import collections
    import string

    # 1. Prepare Corpus and calculate counts to avoid NameError
    corpus_texts = [
        ".",
        "I am learning programming, data structures, and machine learning during my
        "I completed my intermediate education in MPC with strong interest in mathe
        "I want to become a software engineer and build innovative technology solu
        "Artificial intelligence and machine learning are transforming modern indus
        "Cybersecurity protects systems from malware, phishing attacks, and cyber t
        "Cloud computing provides scalable infrastructure and distributed storage s
        "Data science combines statistics, programming, and domain expertise.",
        "Blockchain ensures transparency and security in digital financial transac
        "Quantum computing uses qubits, superposition, and entanglement principles
    ]

    combined_text = " ".join(corpus_texts).lower()
    for p in string.punctuation:
        combined_text = combined_text.replace(p, "")
    words = combined_text.split()

    unigram_counts = collections.Counter(words)
    bigram_counts = collections.Counter(zip(words, words[1:]))
    V = len(unigram_counts)
```

```
def predict_next_word_laplace(word_sequence, bigram_counts, unigram_counts, voc
    words_in_seq = word_sequence.lower().split()
    if not words_in_seq:
        return "Please provide a word sequence."

    last_word = words_in_seq[-1]
    best_word = None
    max_prob = -1

    last_word_count = unigram_counts.get(last_word, 0)

    for candidate in unigram_counts.keys():
        bg_count = bigram_counts.get((last_word, candidate), 0)
        # Laplace formula: P(w2 | w1) = (count(w1, w2) + 1) / (count(w1) + V)
        probability = (bg_count + 1) / (last_word_count + vocabulary_size)

        if probability > max_prob:
            max_prob = probability
            best_word = candidate

    return best_word, max_prob

# Test the smoothed model
test_word = "machine"
predicted, prob = predict_next_word_laplace(test_word, bigram_counts, unigram_c

print(f"Next Word Prediction with Laplace Smoothening")
print(f"------------------------------------------")
print(f"Input word: '{test_word}'")
print(f"Predicted word: '{predicted}'")
print(f"Smoothed Probability: {prob:.4f}")
```

```
Next Word Prediction with Laplace Smoothening
------------------------------------------
Input word: 'machine'
Predicted word: 'learning'
Smoothed Probability: 0.0405
```

Deployment of Laplace Smoothening based Bi-Gram Model

```
import string

# Re-using the logic from the previously executed cell
def deploy_laplace_model():
    if 'predict_next_word_laplace' not in globals():
        print("Error: Please run the previous cell containing the predict_next_
        return

    user_input = input("Enter a word or sequence for prediction: ").strip()

    if not user_input:
        print("Please enter at least one word.")
        return

    # Clean the input to match corpus processing
    clean_seq = user_input.lower()
    for p in string.punctuation:
        clean_seq = clean_seq.replace(p, "")

    predicted_word, probability = predict_next_word_laplace(clean_seq, bigram_c

    print(f"\nInput: '{user_input}'")
    print(f"Predicted next word: '{predicted_word}'")
    print(f"Smoothed Probability: {probability:.4f}")

deploy_laplace_model()
```

```
Enter a word or sequence for prediction: i am

Input: 'i am'
Predicted next word: 'learning'
Smoothed Probability: 0.0274
```

Next Word Prediction Using Tri-Gram Counts based on laplace smoothening

```
import collections
import string

# 1. Prepare data structures
# We need trigrams, bigrams, and the vocabulary size V
V = len(unigram_counts)

def predict_next_word_trigram_laplace(input_text, trigram_counts, bigram_counts
    words = input_text.lower().split()
    if len(words) < 2:
        return "Please enter at least two words.", 0

    w1, w2 = words[-2], words[-1]
    best_word = None
    max_prob = -1

    # Preceding bigram count for the denominator
    bigram_count = bigram_counts.get((w1, w2), 0)

    # Iterate through all words in vocabulary to find the best candidate
    for candidate in unigram_counts.keys():
        # Get trigram count (w1, w2, candidate)
        tg_count = trigram_counts.get((w1, w2, candidate), 0)

        # Laplace Smoothened Probability Formula:
        # P(w3 | w1, w2) = (count(w1, w2, w3) + 1) / (count(w1, w2) + V)
        probability = (tg_count + 1) / (bigram_count + vocabulary_size)

        if probability > max_prob:
            max_prob = probability
            best_word = candidate

    return best_word, max_prob

# Test with a sequence from the corpus
test_seq = "and machine"
predicted_tri, prob_tri = predict_next_word_trigram_laplace(test_seq, trigram_c

print(f"Tri-Gram Prediction with Laplace Smoothening")
print(f"-------------------------------------------")
print(f"Input sequence: '{test_seq}'")
print(f"Predicted next word: '{predicted_tri}'")
print(f"Smoothed Probability: {prob_tri:.4f}")
```

```
Tri-Gram Prediction with Laplace Smoothening
--------------------------------------------
Input sequence: 'and machine'
Predicted next word: 'learning'
Smoothed Probability: 0.0405
```

Deployment of Laplace Smoothening based Tri-Gram Model

```
import string

def deploy_trigram_laplace():
    if 'predict_next_word_trigram_laplace' not in globals():
        print("Error: Please run the previous cell containing the 'predict_nex
        return

    user_input = input("Enter at least two words for Tri-Gram prediction: ").s

    # Basic validation and cleaning
    words = user_input.lower().split()
    if len(words) < 2:
        print("Please enter at least two words for a tri-gram prediction.")
        return

    # Clean the input words
    cleaned_words = []
    for w in words:
        cleaned_w = w.translate(str.maketrans('', '', string.punctuation))
        if cleaned_w: cleaned_words.append(cleaned_w)

    input_sequence = " ".join(cleaned_words)

    predicted_word, probability = predict_next_word_trigram_laplace(
        input_sequence, trigram_counts, bigram_counts, unigram_counts, V
    )

    print(f"\nInput Sequence: '{user_input}'")
    print(f"Predicted next word: '{predicted_word}'")
    print(f"Smoothed Probability: {probability:.4f}")

deploy_trigram_laplace()
```

```
Enter at least two words for Tri-Gram prediction: i am

Input Sequence: 'i am'
Predicted next word: 'learning'
Smoothed Probability: 0.0274
```

Next Word Prediction Using Bi-Gram Counts with Add - K Smoothening

```
import collections
import string

def predict_next_word_add_k(word_sequence, bigram_counts, unigram_counts, vocab
    words_in_seq = word_sequence.lower().split()
    if not words_in_seq:
        return "Please provide a word sequence.", 0

    last_word = words_in_seq[-1]
    best_word = None
    max_prob = -1

    last_word_count = unigram_counts.get(last_word, 0)

    # Add-K formula: P(w2 | w1) = (count(w1, w2) + k) / (count(w1) + k * V)
    for candidate in unigram_counts.keys():
        bg_count = bigram_counts.get((last_word, candidate), 0)
        probability = (bg_count + k) / (last_word_count + k * vocabulary_size)

        if probability > max_prob:
            max_prob = probability
            best_word = candidate

    return best_word, max_prob

# Configuration
K_VALUE = 0.1
V = len(unigram_counts)
test_word = "data"

predicted, prob = predict_next_word_add_k(test_word, bigram_counts, unigram_cou

print(f"Next Word Prediction with Add-K Smoothening (k={K_VALUE})")
print(f"--------------------------------------------------------")
print(f"Input word: '{test_word}'")
print(f"Predicted word: '{predicted}'")
print(f"Smoothed Probability: {prob:.4f}")
```

```
Next Word Prediction with Add-K Smoothening (k=0.1)
------------------------------------------------------
Input word: 'data'
Predicted word: 'structures'
Smoothed Probability: 0.1196
```

Deployment of Add-K Smoothening based Bi-Gram Model

```
import string

def deploy_add_k_model():
    if 'predict_next_word_add_k' not in globals():
        print("Error: Please run the previous cell containing the 'predict_next
        return

    user_input = input("Enter a word for Add-K Bi-Gram prediction: ").strip()
    if not user_input:
        print("Please enter a word.")
```

```
            return

        # Clean input
        clean_seq = user_input.lower().translate(str.maketrans('', '', string.punc

        # Use the same K_VALUE or specify a new one
        k = globals().get('K_VALUE', 0.1)

        predicted_word, probability = predict_next_word_add_k(clean_seq, bigram_co

        print(f"\nInput: '{user_input}' (k={k})")
        print(f"Predicted next word: '{predicted_word}'")
        print(f"Smoothed Probability: {probability:.4f}")

    deploy_add_k_model()
```

```
Enter a word for Add-K Bi-Gram prediction: I

Input: 'I' (k=0.1)
Predicted next word: 'am'
Smoothed Probability: 0.1078
```

Next Word Prediction Using Tri-Gram Counts with Add - K Smoothening

```
import collections
import string

def predict_next_word_trigram_add_k(input_text, trigram_counts, bigram_counts,
    words = input_text.lower().split()
    if len(words) < 2:
        return "Please enter at least two words.", 0

    w1, w2 = words[-2], words[-1]
    best_word = None
    max_prob = -1

    # Preceding bigram count for the denominator
    bigram_count = bigram_counts.get((w1, w2), 0)

    # Iterate through vocabulary to find the best candidate
    for candidate in unigram_counts.keys():
        tg_count = trigram_counts.get((w1, w2, candidate), 0)

        # Add-K Smoothened Probability Formula:
        # P(w3 | w1, w2) = (count(w1, w2, w3) + k) / (count(w1, w2) + k * V)
        probability = (tg_count + k) / (bigram_count + k * vocabulary_size)

        if probability > max_prob:
            max_prob = probability
            best_word = candidate

    return best_word, max_prob

# Configuration
K_VAL_TRI = 0.1
V = len(unigram_counts)
test_seq_tri = "and machine"
```

```
predicted_tri_k, prob_tri_k = predict_next_word_trigram_add_k(
    test_seq_tri, trigram_counts, bigram_counts, unigram_counts, V, k=K_VAL_TR
)

print(f"Tri-Gram Prediction with Add-K Smoothening (k={K_VAL_TRI})")
print(f"---------------------------------------------------")
print(f"Input sequence: '{test_seq_tri}'")
print(f"Predicted next word: '{predicted_tri_k}'")
print(f"Smoothed Probability: {prob_tri_k:.4f}")
```

```
Tri-Gram Prediction with Add-K Smoothening (k=0.1)
---------------------------------------------------
Input sequence: 'and machine'
Predicted next word: 'learning'
Smoothed Probability: 0.2283
```

Deployment of Add-k Smoothening Using Tri-Gram Counts

```
import string

def deploy_trigram_add_k():
    if 'predict_next_word_trigram_add_k' not in globals():
        print("Error: Please run the previous cell containing the 'predict_nex
        return

    user_input = input("Enter at least two words for Add-K Tri-Gram prediction

    # Basic validation
    words = user_input.lower().split()
    if len(words) < 2:
        print("Please enter at least two words.")
        return

    # Clean the input sequence
    cleaned_words = [w.translate(str.maketrans('', '', string.punctuation)) fo
    input_sequence = " ".join([w for w in cleaned_words if w])

    # Retrieve global configuration
    k = globals().get('K_VAL_TRI', 0.1)
    v_size = globals().get('V', len(unigram_counts))

    predicted_word, probability = predict_next_word_trigram_add_k(
        input_sequence, trigram_counts, bigram_counts, unigram_counts, v_size,
    )

    print(f"\nInput Sequence: '{user_input}' (k={k})")
    print(f"Predicted next word: '{predicted_word}'")
    print(f"Smoothed Probability: {probability:.4f}")

deploy_trigram_add_k()
```

```
Enter at least two words for Add-K Tri-Gram prediction: i am

Input Sequence: 'i am' (k=0.1)
Predicted next word: 'learning'
```

```
    Smoothed Probability: 0.1341
```

Thank You