CORPUS

```
text='''
D1:
Artificial intelligence is transforming global industries at an unprecedented velocity.

D2:
Recent developments in quantum cryptography and blockchain ledger technologies are redefining cybersecurity frameworks.

D3:
The metaverse ecosystem integrates augmented reality, neural networks, and decentralized finance protocols.

D4:
Researchers at international laboratories are studying neuroplasticity and bioinformatics to enhance predictive analytics.

D5:
Autonomous vehicles utilize advanced sensor fusion mechanisms to navigate complex urban landscapes.

D6:
Deep learning systems for natural language processing are achieving human-level comprehension in various linguistic tasks.

D7:
The adoption of edge computing paradigms is enabling real-time data processing closer to the source.

D8:
Edge computing reduces latency in distributed systems.

D9:
Edge computing improves efficiency in IoT deployments.

D10:
Advanced AI ecosystems are integrating multiple emerging technologies to drive digital transformation.'''
```

Uni Gram Counts

```
import collections

# Use the 'text' variable from the previously loaded corpus
words = text.lower().split()

# Calculate unigram counts
unigram_counts = collections.Counter(words)

# Print the unigram counts
print("Unigram Counts:")
for word, count in unigram_counts.most_common():
    print(f"{word}: {count}")
#Vocabulary size is length of unigrams
V=len(unigram_counts)
print("Vocabulary Size=",V)
```

Bi-gram count

```
import collections


# Use the 'text' variable from the previously loaded corpus
words = text.lower().split()

# Generate bigrams
bigrams = []
for i in range(len(words) - 1):
    bigrams.append((words[i], words[i+1]))

# Calculate bigram counts
bigram_counts = collections.Counter(bigrams)

# Print the bigram counts
print("\nBigram Counts:")
for bigram, count in bigram_counts.most_common():
    print(f"{bigram[0]} {bigram[1]}: {count}")
```

Tri-gram count

```
import collections


# Use the 'text' variable from the previously loaded corpus
words = text.lower().split()

# Generate trigrams
Trigrams = []
for i in range(len(words) - 2):
    Trigrams.append((words[i], words[i+1], words[i+2]))

# Calculate trigram counts
Trigrams_counts = collections.Counter(Trigrams)

# Print the trigram counts
print("\nTrigrams Counts:")
for Trigrams, count in Trigrams_counts.most_common():
    print(f"{Trigrams[0]} {Trigrams[1]} {Trigrams[2]}: {count}")
```

```
Trigrams Counts:
d1: artificial intelligence: 1
artificial intelligence is: 1
intelligence is transforming: 1
is transforming global: 1
transforming global industries: 1
global industries at: 1
industries at an: 1
at an unprecedented: 1
an unprecedented velocity.: 1
unprecedented velocity. d2:: 1
velocity. d2: recent: 1
d2: recent developments: 1
recent developments in: 1
developments in quantum: 1
in quantum cryptography: 1
quantum cryptography and: 1
cryptography and blockchain: 1
and blockchain ledger: 1
blockchain ledger technologies: 1
ledger technologies are: 1
technologies are redefining: 1
are redefining cybersecurity: 1
redefining cybersecurity frameworks.: 1
cybersecurity frameworks. d3:: 1
frameworks. d3: the: 1
```

```
d3: the metaverse: 1
the metaverse ecosystem: 1
metaverse ecosystem integrates: 1
ecosystem integrates augmented: 1
integrates augmented reality,: 1
augmented reality, neural: 1
reality, neural networks,: 1
neural networks, and: 1
networks, and decentralized: 1
and decentralized finance: 1
decentralized finance protocols.: 1
finance protocols. d4:: 1
protocols. d4: researchers: 1
d4: researchers at: 1
researchers at international: 1
at international laboratories: 1
international laboratories are: 1
laboratories are studying: 1
are studying neuroplasticity: 1
studying neuroplasticity and: 1
neuroplasticity and bioinformatics: 1
and bioinformatics to: 1
bioinformatics to enhance: 1
to enhance predictive: 1
enhance predictive analytics.: 1
predictive analytics. d5:: 1
analytics. d5: autonomous: 1
d5: autonomous vehicles: 1
autonomous vehicles utilize: 1
vehicles utilize advanced: 1
utilize advanced sensor: 1
```

Next Word Prediction Using Bi-Gram Counts

```
def predict_next_word_bigram(word_sequence, bigram_counts, unigram_counts):
    # Tokenize the input sequence and get the last word
    words_in_sequence = word_sequence.lower().split()
    if not words_in_sequence:
        return "Please provide a word sequence."
    last_word = words_in_sequence[-1]

    # Find potential next words based on bigrams starting with last_word
    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}'."

    # Calculate probabilities for potential next words
    # P(w2 | w1) = Count(w1, w2) / Count(w1)
    last_word_unigram_count = unigram_counts.get(last_word, 0)
    if last_word_unigram_count == 0:
        return f"'{last_word}' not found in unigram counts. Cannot predict next word."

    predicted_word = None
    max_probability = -1

    for next_word, bigram_count in potential_next_words.items():
        probability = bigram_count / last_word_unigram_count
        print("probability of  " f'{next_word}' " is ", probability)
        if probability > max_probability:
            max_probability = probability
            predicted_word = next_word

    return predicted_word

# Example usage:
# Ensure bigram_counts and unigram_counts are defined from previous cells
# (They are present in the kernel state.)

# Try with a word sequence
sequence1 = "artificial intelligence"
next_word1 = predict_next_word_bigram(sequence1, bigram_counts, unigram_counts)
print(f"Given sequence: '{sequence1}', predicted next word: '{next_word1}'")

sequence2 = "edge computing"
next_word2 = predict_next_word_bigram(sequence2, bigram_counts, unigram_counts)
print(f"Given sequence: '{sequence2}', predicted next word: '{next_word2}'")
```

```
print(f"Given sequence: '{sequence2}', predicted next word: '{next_word2}'")

sequence3 = "deep learning"
next_word3 = predict_next_word_bigram(sequence3, bigram_counts, unigram_counts)
print(f"Given sequence: '{sequence3}', predicted next word: '{next_word3}'")

sequence4 = "nonexistent word"
next_word4 = predict_next_word_bigram(sequence4, bigram_counts, unigram_counts)
print(f"Given sequence: '{sequence4}', predicted next word: '{next_word4}'")
```

eployment 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}'")
```

Next Word Prediction Using Tri-Gram Counts

```
def predict_next_word_trigram(word_sequence, Trigrams_counts, bigram_counts):
    # Tokenize the input sequence
    words_in_sequence = word_sequence.lower().split()
    if not words_in_sequence:
        return "Please provide a word sequence."

    # Ensure at least two words for trigram prediction
    if len(words_in_sequence) < 2:
        return "Sequence must contain at least two words for trigram prediction."

    # Get the last two words as a tuple
    last_two_words_tuple = tuple(words_in_sequence[-2:])

    # Find potential next words based on trigrams starting with the last two words
    potential_next_words = {}
    for (w1, w2, w3), count in Trigrams_counts.items():
        if (w1, w2) == last_two_words_tuple:
            potential_next_words[w3] = count

    if not potential_next_words:
        return f"No trigram found starting with '{' '.join(last_two_words_tuple)}'."

    # Calculate probabilities for potential next words
    # P(w3 | w1,w2) = Count(w1, w2, w3) / Count(w1, w2)
    # The denominator should be the count of the bigram (w1, w2)
    last_two_words_bigram_count = bigram_counts.get(last_two_words_tuple, 0)
    if last_two_words_bigram_count == 0:
        return f"'{' '.join(last_two_words_tuple)}' not found as a bigram. Cannot predict next word."

    predicted_word = None
    max_probability = -1

    for next_word, trigram_count in potential_next_words.items():
        probability = trigram_count / last_two_words_bigram_count
        print("probability of  " f'{next_word}' " is ", probability)
        if probability > max_probability:
            max_probability = probability
            predicted_word = next_word

    return predicted_word

# Example usage:
# Ensure bigram_counts, unigram_counts, and Trigrams_counts are defined from previous cells
# (They are present in the kernel state.)

# Try with a word sequence
sequence1 = "I am working"
next_word1 = predict_next_word_trigram(sequence1, Trigrams_counts, bigram_counts)
```

```
print(f"Given sequence: '{sequence1}', predicted next word: '{next_word1}'")

sequence2 = "I did my"
next_word2 = predict_next_word_trigram(sequence2, Trigrams_counts, bigram_counts)
print(f"Given sequence: '{sequence2}', predicted next word: '{next_word2}'")
```

Deployment of Tri-Gram Model

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

Next Word Prediction Using Bi-Gram Counts with Laplace Smoothening

✦ Gemini

```
import collections

# Assuming 'text' variable is available globally or needs to be re-defined for self-containment
# For this fix, we'll assume text is defined or re-define it if necessary
# Based on the original notebook, text is defined in F9TVWigHbP47
text='''
D1:
Artificial intelligence is transforming global industries at an unprecedented velocity.

D2:
Recent developments in quantum cryptography and blockchain ledger technologies are redefining cybersecurity frameworks.

D3:
The metaverse ecosystem integrates augmented reality, neural networks, and decentralized finance protocols.

D4:
Researchers at international laboratories are studying neuroplasticity and bioinformatics to enhance predictive analytics.

D5:
Autonomous vehicles utilize advanced sensor fusion mechanisms to navigate complex urban landscapes.

D6:
Deep learning systems for natural language processing are achieving human-level comprehension in various linguistic tasks.

D7:
The adoption of edge computing paradigms is enabling real-time data processing closer to the source.

D8:
Edge computing reduces latency in distributed systems.

D9:
Edge computing improves efficiency in IoT deployments.

D10:
Advanced AI ecosystems are integrating multiple emerging technologies to drive digital transformation.'''

words = text.lower().split()

# Calculate unigram counts (from S-chdArSZaMF)
unigram_counts = collections.Counter(words)
V = len(unigram_counts) # Vocabulary size

# Generate bigrams and calculate bigram counts (from PgvPd-z-Zuf0)
bigrams = []
for i in range(len(words) - 1):
    bigrams.append((words[i], words[i+1]))
bigram_counts = collections.Counter(bigrams)

def predict_next_word_bigram_Laplace(word_sequence, bigram_counts, unigram_counts):
    # Tokenize the input sequence and get the last word
    words_in_sequence = word_sequence.lower().split()
    if not words_in_sequence:
        return
```

Deployment of Laplace Smoothening based Bi-Gram Model

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

```
enter text
```

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

```
def predict_next_word_trigram_Laplace(word_sequence, Trigrams_counts, bigram_counts):
    # Tokenize the input sequence
    words_in_sequence = word_sequence.lower().split()
    if not words_in_sequence:
        return "Please provide a word sequence."

    # Ensure at least two words for trigram prediction
    if len(words_in_sequence) < 2:
        return "Sequence must contain at least two words for trigram prediction."

    # Get the last two words as a tuple
    last_two_words_tuple = tuple(words_in_sequence[-2:])

    # Find potential next words based on trigrams starting with the last two words
    potential_next_words = {}
    for (w1, w2, w3), count in Trigrams_counts.items():
        if (w1, w2) == last_two_words_tuple:
            potential_next_words[w3] = count

    if not potential_next_words:
        return f"No trigram found starting with '{' '.join(last_two_words_tuple)}'."

    # Calculate probabilities for potential next words
    # P(w3 | w1,w2) = Count(w1, w2, w3) / Count(w1, w2)
    # The denominator should be the count of the bigram (w1, w2)
    last_two_words_bigram_count = bigram_counts.get(last_two_words_tuple, 0)
    if last_two_words_bigram_count == 0:
        return f"'{' '.join(last_two_words_tuple)}' not found as a bigram. Cannot predict next word."

    predicted_word = None
    max_probability = -1

    for next_word, trigram_count in potential_next_words.items():
        probability = (trigram_count+1) / (last_two_words_bigram_count+V)
        print("probability of  " f'{next_word}' " is ", probability)
        if probability > max_probability:
            max_probability = probability
            predicted_word = next_word

    return predicted_word

# Example usage:
# Ensure bigram_counts, unigram_counts, and Trigrams_counts are defined from previous cells
# (They are present in the kernel state.)

# Try with a word sequence
sequence1 = "I am working"
next_word1 = predict_next_word_trigram_Laplace(sequence1, Trigrams_counts, bigram_counts)
print(f"Given sequence: '{sequence1}', predicted next word: '{next_word1}'")

sequence2 = "I did my"
```

```
next_word2 = predict_next_word_trigram_Laplace(sequence2, Trigrams_counts, bigram_counts)
print(f"Given sequence: '{sequence2}', predicted next word: '{next_word2}'")
```

```
probability of  as is  0.11764705882352941
probability of  in is  0.11764705882352941
Given sequence: 'I am working', predicted next word: 'as'
probability of  phd is  0.11764705882352941
probability of  masters is  0.11764705882352941
Given sequence: 'I did my', predicted next word: 'phd'
```

Deployment of Laplace Smoothening based Tri-Gram Model

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

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

```
def predict_next_word_bigram_K(word_sequence, bigram_counts, unigram_counts, K): #K=0.5-0.01
    # Tokenize the input sequence and get the last word
    words_in_sequence = word_sequence.lower().split()
    if not words_in_sequence:
        return "Please provide a word sequence."
    last_word = words_in_sequence[-1]

    # Find potential next words based on bigrams starting with last_word
    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}'."

    # Calculate probabilities for potential next words
    # P(w2 | w1) = Count(w1, w2) / Count(w1)
    last_word_unigram_count = unigram_counts.get(last_word, 0)
    if last_word_unigram_count == 0:
        return f"'{last_word}' not found in unigram counts. Cannot predict next word."

    predicted_word = None
    max_probability = -1

    for next_word, bigram_count in potential_next_words.items():
        probability = (bigram_count+K) / (last_word_unigram_count+K*V)
        print("probability of " f'{next_word}' " is ", probability)
        if probability > max_probability:
            max_probability = probability
            predicted_word = next_word

    return predicted_word

# Example usage:
# Ensure bigram_counts and unigram_counts are defined from previous cells
# (They are present in the kernel state.)

# Try with a word sequence
sequence1 = "I am"
next_word1 = predict_next_word_bigram_K(sequence1, bigram_counts, unigram_counts,0.5)
print(f"Given sequence: '{sequence1}', predicted next word: '{next_word1}'")

sequence2 = "I did my"
next_word2 = predict_next_word_bigram_K(sequence2, bigram_counts, unigram_counts,0.5)
print(f"Given sequence: '{sequence2}', predicted next word: '{next_word2}'")

sequence3 = "professor I"
next_word3 = predict_next_word_bigram_K(sequence3, bigram_counts, unigram_counts,0.5)
print(f"Given sequence: '{sequence3}', predicted next word: '{next_word3}'")

sequence4 = "nonexistent word"
next_word4 = predict_next_word_bigram_K(sequence4, bigram_counts, unigram_counts,0.5)
print(f"Given sequence: '{sequence4}', predicted next word: '{next_word4}'")
```

Deployment of Add-K Smoothening based Bi-Gram Model

```
ip_text=input("enter text")
next_word1 = predict_next_word_bigram_K(ip_text, bigram_counts, unigram_counts,0.5)
print(f"Given sequence: '{ip_text}', predicted next word: '{next_word1}'")
```

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

```
def predict_next_word_trigram_K(word_sequence, Trigrams_counts, bigram_counts,K): #K=0.5-0.01
    # Tokenize the input sequence
    words_in_sequence = word_sequence.lower().split()
    if not words_in_sequence:
        return "Please provide a word sequence."

    # Ensure at least two words for trigram prediction
    if len(words_in_sequence) < 2:
        return "Sequence must contain at least two words for trigram prediction."

    # Get the last two words as a tuple
    last_two_words_tuple = tuple(words_in_sequence[-2:])

    # Find potential next words based on trigrams starting with the last two words
    potential_next_words = {}
    for (w1, w2, w3), count in Trigrams_counts.items():
        if (w1, w2) == last_two_words_tuple:
            potential_next_words[w3] = count

    if not potential_next_words:
        return f"No trigram found starting with '{' '.join(last_two_words_tuple)}'."

    # Calculate probabilities for potential next words
    # P(w3 | w1,w2) = Count(w1, w2, w3) / Count(w1, w2)
    # The denominator should be the count of the bigram (w1, w2)
    last_two_words_bigram_count = bigram_counts.get(last_two_words_tuple, 0)
    if last_two_words_bigram_count == 0:
        return f"'{' '.join(last_two_words_tuple)}' not found as a bigram. Cannot predict next word."

    predicted_word = None
    max_probability = -1

    for next_word, trigram_count in potential_next_words.items():
        probability = (trigram_count+K) / (last_two_words_bigram_count+K*V)
        print("probability of  " f'{next_word}' " is ", probability)
        if probability > max_probability:
            max_probability = probability
            predicted_word = next_word

    return predicted_word

# Example usage:
# Ensure bigram_counts, unigram_counts, and Trigrams_counts are defined from previous cells
# (They are present in the kernel state.)

# Try with a word sequence
sequence1 = "I am working"
next_word1 = predict_next_word_trigram_K(sequence1, Trigrams_counts, bigram_counts,0.5)
print(f"Given sequence: '{sequence1}', predicted next word: '{next_word1}'")

sequence2 = "I did my"
next_word2 = predict_next_word_trigram_K(sequence2, Trigrams_counts, bigram_counts,0.5)
print(f"Given sequence: '{sequence2}', predicted next word: '{next_word2}'")
```

Deployment of Add-K Smoothening based Tri-Gram Model

```
ip_text=input("enter text")
next_word1 = predict_next_word_trigram_K(ip_text, Trigrams_counts, bigram_counts,0.5)
print(f"Given sequence: '{ip_text}', predicted next word: '{next_word1}'")
```