Understanding N-grams

Minor in AI - IIT ROPAR

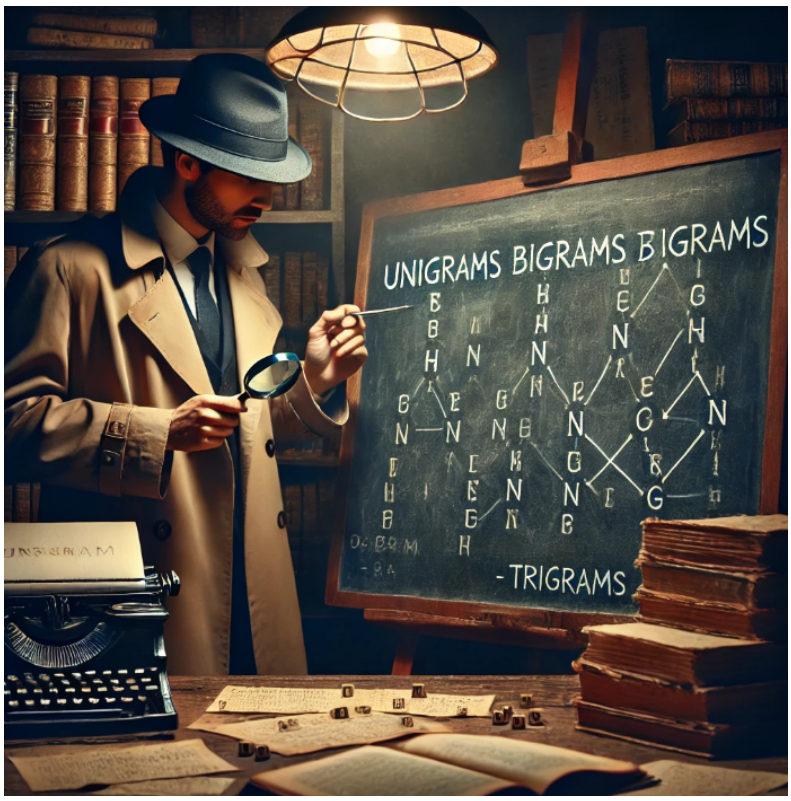
5th March, 2025

The Cryptic Manuscript: Unraveling Hidden Patterns in Ancient Texts

Imagine you are a detective trying to decipher a mysterious message hidden within an ancient manuscript. The manuscript is filled with thousands of words, and your task is to uncover patterns in how these words appear together. This is no ordinary text—it holds secrets that can only be revealed by carefully analyzing its structure.

To break down the message, you start by examining individual words, counting how often each appears. This method is called a unigram analysis. But a single word alone may not reveal much, so you move to the next step—looking at pairs of words that frequently appear together. These are called bigrams, and they help you understand relationships between words. Taking it a step further, you begin analyzing sequences of three words, known as trigrams, to see if a deeper pattern emerges.

By studying the frequency and order of these word sequences, you can predict which words are likely to follow others. This technique is widely used in text analysis, machine translation, and speech recognition. Whether you are deciphering an ancient manuscript or training an AI to understand human language, recognizing these hidden structures brings you one step closer to unlocking the true meaning of the text.



What is an N-Gram?

An **N-Gram** is a continuous sequence of N items (words or characters) from a given text. In Natural Language Processing (NLP), **N-Grams** help analyze and understand patterns in text, which is useful for *text prediction, speech recognition, machine translation, sentiment analysis*, and many other applications.

Breaking Down N-Grams with an Example

Consider the sentence:

> "I love machine learning"

- **Unigrams** (N = 1): {I, love, machine, learning}
- **Bigrams** (N = 2): {I love, love machine, machine learning}
- **Trigrams** (N = 3): {I love machine, love machine learning}
- **4-Grams** (N = 4): {I love machine learning}

If N increases further, we get longer phrases, but they appear less frequently in real text.

Why are N-Grams Useful?

Text Prediction (Auto-Complete & Spell Correction)

- If you type "How are", the model suggests "you?" based on frequent bigrams.
- If a sentence has a typo, the model predicts the correct word using common N-Grams.

Speech Recognition

If you say "I need a glass of...", the system predicts "water" based on common trigrams.

Machine Translation

Instead of translating word-by-word, modern translation systems use N-Grams to understand sentence structure.

Sentiment Analysis

Identifying common bigrams and trigrams helps detect sentiment:

- "not happy" (bigram) and "really bad experience" (trigram) often indicate negativity.

Text Classification & Spam Detection

Certain N-Grams are common in spam messages, e.g., "Win a free iPhone" (trigram). Spam filters use N-Grams to identify and block unwanted emails.

Types of N-Grams

Word N-Grams

Definition: A sequence of N words.
Example (N = 3): "The quick brown fox" $\rightarrow$ {The quick brown, quick brown fox}.

Character N-Grams

Definition: A sequence of N characters (ignoring word boundaries).
Example (N = 3) for "hello": {hel, ell, llo}.

N-Gram Type	Pros	Cons
Unigrams (N=1)	Simple, useful for bag-of-words models	Lacks context
Bigrams (N=2)	Captures some word relationships	Doesn't capture long-term dependencies
Trigrams (N=3)	Understands phrases better	Needs more data to be effective
4-Grams & Above	Very context-rich	Requires huge amounts of text and computation

Choosing the Right N for N-Grams

Probabilities in N-Grams (Markov Assumption)

The probability of a word appearing is given by:

$$P(w_n|w_{n-1}) = \frac{\text{count}(w_{n-1}, w_n)}{\text{count}(w_{n-1})} \tag{1}$$

For example: If "machine learning" appears 50 times, and "machine" appears 200 times, then:

$$P(\text{learning}|\text{machine}) = \frac{50}{200} = 0.25 \tag{2}$$

Using trigram probabilities:

$$P(w_n|w_{n-2}, w_{n-1}) = \frac{\text{count}(w_{n-2}, w_{n-1}, w_n)}{\text{count}(w_{n-2}, w_{n-1})} \tag{3}$$

Now, let's break down the Python code that performs N-gram analysis on H.G. Wells' novel The Time Machine step by step.

Step 1: Reading and Preprocessing the Text

```
with open("The_Time_Machine.txt", "r", encoding="utf-8") as f:
    lines = f.read()
```

We open the text file in read mode with UTF-8 encoding to ensure all characters are read correctly.

Step 2: Converting Text to Lowercase and Removing Punctuation

```
import re

lines = lines.lower()
lines = re.sub(r"[^a-z\s]", "", lines)
```

- Converts text to lowercase so that "Time" and "time" are treated the same.
- Removes punctuation, numbers, and special characters, keeping only lowercase letters and spaces.

Step 3: Tokenization and Vocabulary Building

```
tokens = lines.split()
vocabulary = set(tokens)
```

- Splits text into words (tokens) by spaces.
- Extracts unique words into a vocabulary set.

Step 4: Counting Word Frequencies

```
from collections import Counter

word_freq = Counter(tokens)

for i in word_freq.most_common(5):
    print(i)
```

- Counts the frequency of each word and prints the top 5 most common words.

Expected Output

('the', 2468)
('and', 1296)
('of', 1281)
('i', 1242)
('a', 864)

Step 5: Sorting Word Frequencies for Visualization

```
sorted_word_freq = dict(sorted(word_freq.items(), key=lambda item: item[1], reverse=True
    ))
word = list(sorted_word_freq.keys())[:50]
freq = list(sorted_word_freq.values())[:50]
```

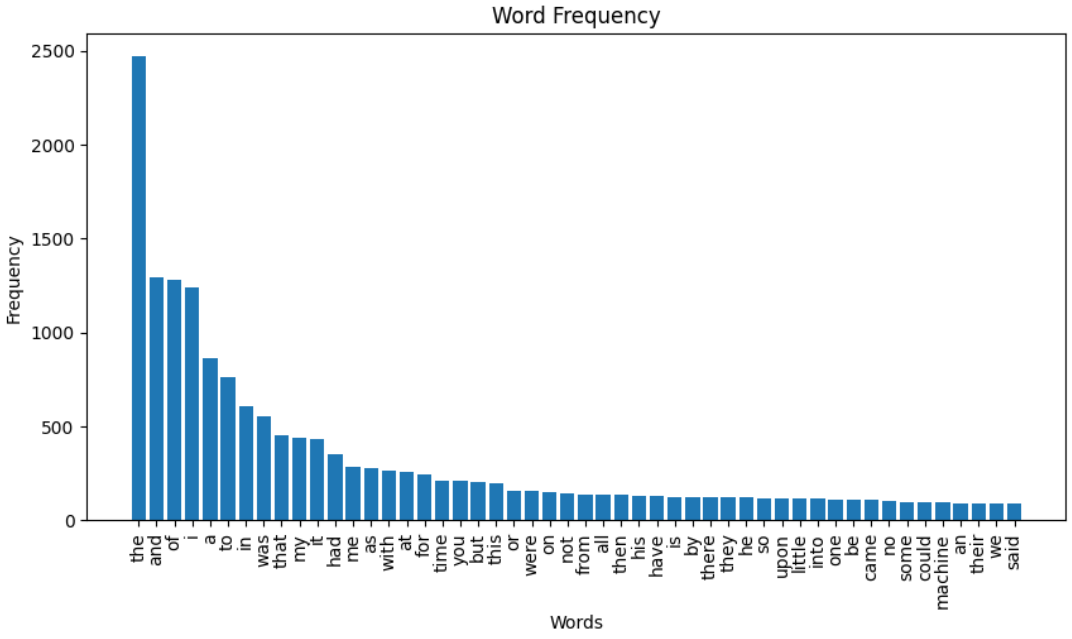
- Sorts the dictionary in descending order of frequency and extracts the top 50 words and their frequencies.

Step 6: Visualizing Word Frequency

```
import matplotlib.pyplot as plt

plt.figure(figsize=(10,5))
plt.bar(word, freq)
plt.xticks(rotation=90)
plt.xlabel("Words")
plt.ylabel("Frequency")
plt.title("Word Frequency")
plt.show()
```

- Generates a bar chart showing the frequency of the 50 most common words.



Step 7: Defining the N-gram Generator

```
def generate_ngrams(tokens, n):
    return [" ".join(tokens[i:i+n]) for i in range(len(tokens)-n+1)]
```

- Generates N-grams as sequences of words instead of concatenated strings.

Step 8: Creating Unigrams, Bigrams, and Trigrams

```
unigram = tokens
bi_gram = generate_ngrams(tokens, 2)
tri_gram = generate_ngrams(tokens, 3)
```

- Stores individual words, two-word sequences, and three-word sequences.

Step 9: Counting N-gram Frequencies

```
unigram_freq = Counter(unigram)
bi_gram_freq = Counter(bi_gram)
tri_gram_freq = Counter(tri_gram)
```

- Counts occurrences of unigrams, bigrams, and trigrams.

Step 10: Sorting N-gram Frequencies

```
unigram_sorted = sorted(unigram_freq.values(), reverse=True)
bi_gram_sorted = sorted(bi_gram_freq.values(), reverse=True)
tri_gram_sorted = sorted(tri_gram_freq.values(), reverse=True)
```

- Sorts frequency values in descending order.

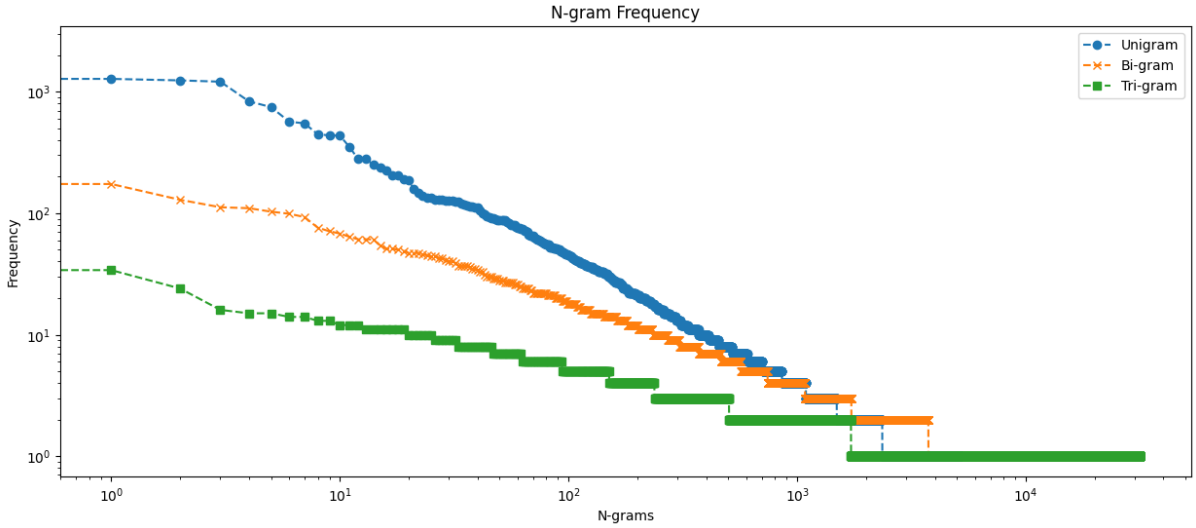
Step 11: Plotting N-gram Frequencies on a Log-Log Scale

```
plt.figure(figsize=(15,6))
plt.plot(unigram_sorted, label="Unigram", marker='o', linestyle='--')
plt.plot(bi_gram_sorted, label="Bi-gram", marker='x', linestyle='--')
plt.plot(tri_gram_sorted, label="Tri-gram", marker='s', linestyle='--')

plt.yscale('log')
plt.xscale('log')

plt.xlabel("N-grams")
plt.ylabel("Frequency")
plt.title("N-gram Frequency")
plt.legend()
plt.show()
```

- Uses a log-log plot since word frequencies follow Zipf's Law, where a few words are common while most are rare.
- Different markers ('o', 'x', 's') represent unigram, bigram, and trigram counts.



Challenges with N-Grams

Data Sparsity

Many trigrams and higher N-Grams occur rarely, making it hard to learn probabilities.
Solution: Smoothing techniques like *Laplace Smoothing* help assign nonzero probabilities to unseen words.

High Memory & Computation Cost

Storing bigram or trigram probabilities for large corpora is expensive.
Solution: Neural language models (e.g., GPT, LSTMs) overcome this by learning word embeddings.

Lack of Long-Range Dependencies

A trigram model can't understand dependencies between words far apart in a sentence.
Solution: Transformers (like GPT-3) use attention mechanisms to capture long-term dependencies.

Key Takeaways

- An **N-Gram** is a sequence of N words appearing together in a text. **Unigrams** are single words, **bigrams** are two-word sequences, and **trigrams** are three-word sequences. Higher-order N-Grams improve contextual understanding but require significantly more data.
- N-Grams are widely used in **text prediction**, such as autocomplete and spell-checking, where bigrams and trigrams help predict the next word. They also play a crucial role in **speech recognition** by suggesting words based on common sequences. In **machine translation**, N-Grams assist in structuring translated sentences more accurately by identifying frequent word combinations.
- In **sentiment analysis**, certain bigrams and trigrams help detect positive or negative sentiment, such as *"not happy"* or *"really bad experience"*. Similarly, in **spam detection**, common spam phrases like *"win a free iPhone"* can be identified using N-Gram analysis.