Data Hunger: Why N-Gram Models Struggle with Sparsity (and What Modern LLMs Do Differently)

TL;DR: *Higher-order n-gram models struggle with the exponential data requirements and sparsity issues of capturing language patterns, while modern LLMs overcome these challenges with neural architectures, embeddings, and sub-word tokenization*.

In this post, we will explore why traditional n-gram models hit a wall as they attempt to use more context, delving into their insatiable appetite for data and the problems caused by sparsity. We'll also contrast these limitations with the breakthroughs of modern large language models (LLMs), which have transformed how we approach language modelling.

Introduction

Imagine you start building a language model with a simple approach: counting how often certain words follow each other. You begin with something very basic - just look at one word at a time (unigram), or maybe pairs of words (bigrams). With a modest amount of text, you can create a model that occasionally produces semi-coherent short fragments. Encouraged by this, you decide to add more context to improve coherence. Instead of just looking at single words or pairs, what if you consider three words or four words at a time?

In the last post, we did just that: we got up to trigrams (three-word sequences) to demonstrate with very little code and a sample text of a million words that plausible language can be generated.

Moving up to 4 or 5 words sounds like a natural step - more context should mean more natural-sounding text, right? The catch is that as you increase the number of words in your "window," you also increase the complexity of your model. Suddenly, the amount of data you need to avoid repetitive, looped, or nonsensical output skyrockets. Without a massive amount of text, your new higher-order model might just keep repeating the same phrases over and over, or struggle to find any valid next words at all.

In this post, we'll explore why going beyond trigrams into tetragrams and beyond requires exponentially more text, what happens when you don't have enough data, and how this connects to the broader world of advanced language models. By the end, you'll

have a clearer picture of how the desire for richer local context leads straight into the arms of ever-growing datasets.

A Quick Refresher on N-Grams

First introduced by Claude Channon in his highly influential paper, "A mathematical theory of communication" (Shannon, 1948), **n-grams** are sequences of n consecutive words (or tokens) extracted from text. For example, if n = 2, we talk about **bigrams**, which are pairs of consecutive words like "natural language" or "machine learning." Similarly, if n = 3, we have **trigrams**, such as "large language models."

In Natural Language Processing (NLP), n-grams are used to build simple statistical models of language. By counting how often certain sequences of words appear, these models can predict the next word in a sentence, generate text, or estimate how "natural" a piece of text sounds. The basic idea is: if you know which words frequently follow one another, you can guess the next word more accurately. This approach doesn't require any understanding of the text's meaning; it's purely based on patterns observed in sample documents.

In natural language, even a very large text corpus can't contain every possible word sequence. Many n-grams will be rare or unseen (Jurafsky & Martin, 2024). A basic frequency-based model would assign zero probability to these unseen n-grams, which isn't very practical. **Smoothing** techniques adjust the raw frequency counts so that the model can assign a small, non-zero probability to unseen or rare n-grams, thus improving generalization and overall model performance. Kneser-Ney smoothing (Kneser & Ney, 1995) became a seminal advancement in how n-gram probabilities are estimated, improving performance over more basic smoothing techniques and becoming a standard baseline method in n-gram language modelling.

Data Requirements Grow Exponentially

When you move from bigrams to trigrams, and then to tetragrams and beyond, you're essentially increasing the "resolution" of your language model. Each additional word in the context window doesn't just add a little complexity, it multiplies the number of possible sequences the model needs to keep track of.

Consider a simple example:

- Suppose you have a vocabulary of V unique words.
- A unigram model just counts how often each single word appears, so it deals with V possible items.

- A bigram model considers pairs of words. In theory, there could be up to V × V = V^2 possible bigrams.
- A trigram model looks at triplets, so that's up to V^3 possible trigrams.

In practice, the size of your vocabulary depends on your dataset and how you process it. If you're dealing with English text and apply some basic normalization (like lowercasing and removing very rare words), you can still easily end up with tens of thousands of unique words. For instance:

- Small, Curated Texts: A focused dataset (e.g., specific domain literature) might have a few thousand unique words.
- General English Text (News, selective Wikipedia): It's common to see 50,000 to 200,000 unique words, even after filtering out extremely rare terms.
- Large, Unfiltered Corpora: With large and diverse datasets (like the entirety of Wikipedia or large web crawls), the raw vocabulary can reach into the millions, including proper nouns, technical terms, and rare words. Many NLP practitioners then apply methods to reduce or manage this vocabulary, such as using sub-word tokenization.

In our previous post, we combined some classics into a single text file for our gentext.py program to use. Let's see how many unique lowercase words were contained in this corpus of just over a million words overall:

```
  11:31:31 User@AN20 ~/Demystifying LLMs 491 0 😊 $ tr '[:space:]' '\n' <
sample.txt | tr '[:upper:]' '[:lower:]' | sort -u | wc -l
  69132
```

So gentext.py was working with V = 69,132 and we moved up to a trigram model making a possible 330 trillion trigrams! Let's see with our million-word corpus how many trigrams were actually created – here's the existing code with some print statements added:

```
# -------------------------
# TRIGRAM MODEL (WORD-BASED)
# -------------------------
trigram_counts = defaultdict(Counter)
for i in range(len(words)-2):
    pair = (words[i], words[i+1])
    next_word = words[i+2]
    trigram_counts[pair][next_word] += 1

# Print some stats about the trigram model
```

```
num_unique_pairs = len(trigram_counts)
num_unique_trigrams = sum(len(cdict) for cdict in trigram_counts.values())
print("Trigram model stats:")
print(f"Number of unique word pairs (contexts): {num_unique_pairs}")
print(f"Number of unique trigrams (contexts + next word): {num_unique_trigrams}")
```

Running this gives us:

```
Trigram model stats:
Number of unique word pairs (contexts): 360680
Number of unique trigrams (contexts + next word): 792356
```

So nearly 800k trigrams from a million-word dataset may indicate richness and diversity but it's still very sparse compared to the full combinatorial space.

Let's try a different dataset. We could use the set of over 200k Jeopardy questions and answers. This is a 53MB JSON file that looks like this:

```
01:10:21 User@AN20 ~/DataHungerOfN-GramModels 421 130 😡 $ head -c 1350
JEOPARDY_QUESTIONS1.json
[{"category": "HISTORY", "air_date": "2004-12-31", "question": "'For the
last 8 years of his life, Galileo was under house arrest for espousing this
man's theory'", "value": "$200", "answer": "Copernicus", "round":
"Jeopardy!", "show_number": "4680"}, {"category": "ESPN's TOP 10 ALL-TIME
ATHLETES", "air_date": "2004-12-31", "question": "'No. 2: 1912 Olympian;
football star at Carlisle Indian School; 6 MLB seasons with the Reds, Giants
& Braves'", "value": "$200", "answer": "Jim Thorpe", "round": "Jeopardy!",
"show_number": "4680"}, {"category": "EVERYBODY TALKS ABOUT IT...",
"air_date": "2004-12-31", "question": "'The city of Yuma in this state has a
record average of 4,055 hours of sunshine each year'", "value": "$200",
"answer": "Arizona", "round": "Jeopardy!", "show_number": "4680"},
{"category": "THE COMPANY LINE", "air_date": "2004-12-31", "question": "'In
1963, live on \"The Art Linkletter Show\", this company served its billionth
burger'", "value": "$200", "answer": "McDonald\\'s", "round": "Jeopardy!",
"show_number": "4680"}, {"category": "EPITAPHS & TRIBUTES", "air_date":
"2004-12-31", "question": "'Signer of the Dec. of Indep., framer of the
Constitution of Mass., second President of the United States'", "value":
"$200", "answer": "John Adams", "round": "Jeopardy!", "show_number": "4680"},
```

It's all on one line and, being a JSON file, has some useful metadata. Let's strip out only the questions and answers using rather brilliant jq:

```
01:08:22 User@AN20 ~/DataHungerOfN-GramModels 302 0 😊 $ jq -r '.[] |
(.question, .answer)' JEOPARDY_QUESTIONS1.json > sample.txt
01:08:34 User@AN20 ~/DataHungerOfN-GramModels 314 0 😊 $ wc -w sample.txt
3572429 sample.txt
01:08:42 User@AN20 ~/DataHungerOfN-GramModels 322 0 😊 $ head sample.txt
'For the last 8 years of his life, Galileo was under house arrest for
espousing this man's theory'
Copernicus
'No. 2: 1912 Olympian; football star at Carlisle Indian School; 6 MLB
seasons with the Reds, Giants & Braves'
Jim Thorpe
'The city of Yuma in this state has a record average of 4,055 hours of
sunshine each year'
Arizona
```

```
   'In 1963, live on "The Art Linkletter Show", this company served its
billionth burger'
   McDonald\'s
   'Signer of the Dec. of Indep., framer of the Constitution of Mass., second
President of the United States'
   John Adams
   01:21:54 User@AN20 ~/DataHungerOfN-GramModels 114 0 😊 $ tr '[:space:]'
'\n' < sample.txt | tr '[:upper:]' '[:lower:]' | sort -u | wc -l
   281792
```

OK, so we now have a 3.5m word count sample.txt with 281,792 unique words which generate...

```
   Trigram model stats:
   Number of unique word pairs (contexts): 1273837
   Number of unique trigrams (contexts + next word): 2508025
```

...2.5m trigrams.

Oh dear, this is even more sparse - V is much larger now and needs to be cubed but the number of trigrams hasn't increased by the same magnitude. Let's put this into numbers by defining sparsity and plugging in the values for the 2 modules:

$$Sparsity = 1 - \frac{Num\ Observed\ NGrams\ of\ V^N}{Total\ Possible\ NGrams\ of\ V^N}$$

The sparsity value ranges between:

- 0 (no sparsity): Every possible n-gram has been observed.
- 1 (complete sparsity): None of the possible n-grams have been observed.

For dataset 1:

$$Sparsity = 1 - \frac{792356}{69132^3} = 1 - \frac{792356}{3.3 \times 10^{14}} \cong 1 - 2.4 \times 10^{9} \approx 1$$

The sparsity is extremely close to 1, meaning only an infinitesimal fraction of all possible trigrams has been observed. And for dataset 2:

$$Sparsity = 1 - \frac{2508025}{281792^3} = 1 - \frac{2508025}{2.24 \times 10^{16}} \cong 1 - 1.12 \times 10^{10} \approx 1$$

Again, the sparsity is overwhelmingly close to 1, even though the dataset is larger and captures more trigrams.

Hitting the Limits With Small Datasets

We saw above how data sparsity rears its head because the model needs enough examples of each n-gram to accurately estimate probabilities. With high sparsity, we would get dead ends, repetitive loops and overfitting.

These issues highlight why early statistical approaches often stopped at trigrams. As soon as you push to tetragrams or more, the data demands skyrocket further, and small to moderate datasets can't keep up. This becomes a hard practical limit, showing why either much larger corpora or more advanced modelling techniques (like smoothing, or ultimately, neural networks) are essential for better language modelling.

As an example of a large dataset, Google's "Web 1T 5-gram" model is an impressive pre-computed n-gram model with the following stats (Franz & Brants 2006):

Tokens	1,024,908,267,229
Sentences	95,119,665,584
Unigrams	13,588,391
Bigrams	314,843,401
Trigrams	977,069,902
Fourgrams	1,313,818,354
Fivegrams	1,176,470,663

Without even doing the calculations, I think it is obvious that even this has the sparsity problem - simply because the vocabulary size (approximately the same as the count of unigrams depending on tokenization) is so large at over 13m, despite generating 977m trigrams from the 1 trillion source!

Comparisons to Advanced Language Models

While higher-order n-gram models illuminate the exponential data requirements for capturing language patterns, modern **large language models (LLMs)** have taken a different route. Instead of explicitly enumerating all possible n-gram sequences, they leverage neural network architectures (e.g., Transformers) that learn continuous representations of words and context. This approach provides several advantages:

- Continuous embeddings vs. discrete N-Grams
    - In an n-gram model, each word is a discrete unit, and we track exact counts of sequences. This can quickly become sparse as n grows.
    - By contrast, LLMs embed words (or more often, sub-word tokens) into high-dimensional vectors. Similar words or phrases end up close to each other in this continuous space, reducing the sparsity problem. The model can generalize to contexts it hasn't seen verbatim, because "similar" contexts map to related areas in the "**embedding space**".
- Massive datasets, but not combinatorial enumeration

- Modern LLMs are trained on massive corpora (often petabytes of data), which dwarf traditional n-gram datasets.
- Rather than counting each n-gram, they learn to approximate the probability distribution over tokens via billions of parameters. They still need huge amounts of data, but their neural architecture lets them reuse and generalize patterns in ways that a straightforward n-gram model cannot.

Advanced language models have outgrown the strict limitations of n-gram enumeration, but they are still, in essence, probabilistic text generators - just far more flexible and powerful.

Let's have a look at two key advancements of LLMs that produce such impressive language processing.

Sub-word Tokenisation and Vocabulary Size

Traditional word-level tokenization requires a massive vocabulary to cover every possible word in a language, resulting in extreme sparsity and inefficiency - especially when dealing with rare words or languages with complex morphology. Sub-word tokenization solves this by breaking words into smaller, reusable units.

Using sub-word units like prefixes, suffixes, and roots drastically reduces the vocabulary size while retaining the ability to reconstruct any word. In traditional word-level models, V can reach hundreds of thousands, especially for large corpora (we saw this in our Jeopardy dataset above). Yet, modern sub-word models (e.g.BERT, GPT-3), V is typically 30,000 to 50,000 tokens, despite being trained on datasets spanning trillions of words.

Furthermore, the same set of sub-words can be used in related languages whereas whole word based LMs would have had separate vocabularies for each language.

Hidden Multi-Dimensional Vectors

At the heart of modern language models lies a fascinating concept: **hidden multi-dimensional vectors**, commonly referred to as **embeddings**. These vectors are dense, high-dimensional numerical representations that encode the properties of tokens (words, sub-words, or characters) in a way that computers can process. They are central to how models like GPT-4, BERT, and others understand and generate text.

We'll explore these in a future post - just as a teaser, I'll mention that GPT-3 represents each sub-word token as a 12,288-dimensional vector, that's a lot of information for each token, of which there are 50,257. Imagine having a 2D array representing this embedding space...

```
float tokens[50257][12288];
```

...this is how PyTorch and TensorFlow implement the embedding table.

Conclusion

Higher-order n-gram models are a fascinating step in the evolution of language modelling, but they hit a wall pretty quickly. They crave data - massive amounts of it - and even then, they struggle with sparsity, which makes them feel a bit like a relic in the age of massive neural networks. Modern language models have sidestepped these problems with clever tricks like embeddings and sub-word tokenization, giving them the flexibility and power that n-grams could only dream of. While n-gram models remain a cornerstone in understanding the evolution of LMs, the advancements in neural networks demonstrate the power of innovation. But hey, n-grams still have their place - they're simple, elegant, and surprisingly useful in the right context.

Postscript - N-Grams Live On

N-gram LMs are far from dead though: the **Infini-gram** model (Liu et al, 2024) represents a significant advancement by extending traditional n-gram models to handle unbounded n values, effectively creating an "∞-gram" model. There's no pre-computing as that would be too onerous on the massive corpus they used - 5 trillion words! Instead **suffix arrays** are used (Manber & Myers, 1993) based on suffix trees (Weiner, 1973). This data structure allows for on-the-fly computation of n-gram probabilities with millisecond-level latency, even for massive datasets.

References

Brantz, T., & Franz, A. (2006). The google web 1T 5-gram corpus. *Linguistic Data Consortium*. UPenn
This corpus was a groundbreaking resource in language modelling, providing a massive dataset of n-grams that enabled researchers to explore higher-order models on an unprecedented scale.

Jurafsky, Daniel and Martin, James H. (2024). "Chapter 3: N-gram Language Models". "Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition with Language Models", 3rd edition. Stanford
This chapter is a foundational introduction to n-gram models and their role in computational linguistics. It succinctly explains the concepts of smoothing, sparsity, and practical limitations, making it an essential resource for anyone studying or working in NLP.

Liu, J., Min, S., Zettlemoyer, L., Choi, Y., & Hajishirzi, H. (2024). Infini-gram: Scaling unbounded n-gram language models to a trillion tokens. *arXiv preprint arXiv:2401.17377*. arXiv
This work pushes the boundaries of traditional n-gram models, demonstrating how innovative data structures and algorithms can scale to unprecedented levels. It showcases the continued relevance of n-grams in the era of neural models and offers insights into bridging classic and modern approaches.

Manber, U., & Myers, G. (1993). Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, *22*(5), 935-948. Max Planck Society
The introduction of suffix arrays was a major milestone in computational string processing. This efficient data structure is widely used in NLP tasks, from n-gram modeling to modern text indexing and retrieval, demonstrating its enduring versatility and impact.

Shannon, C. E. (1948). A mathematical theory of communication. *The Bell system technical journal*, *27*(3), 379-423. Max Planck Society
Shannon's paper laid the theoretical groundwork for information theory and language modeling. His introduction of the n-gram concept and probabilistic frameworks continues to influence how we understand and model language and information processing.

Weiner, P. (1973, October). Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory (swat 1973)* (pp. 1-11). IEEE. Yale
Weiner's work on suffix trees provided an efficient way to manage and search large datasets, which has been instrumental in text processing, indexing, and n-gram computation. It remains a cornerstone of algorithms used in computational linguistics and beyond.