Foundations of NLP

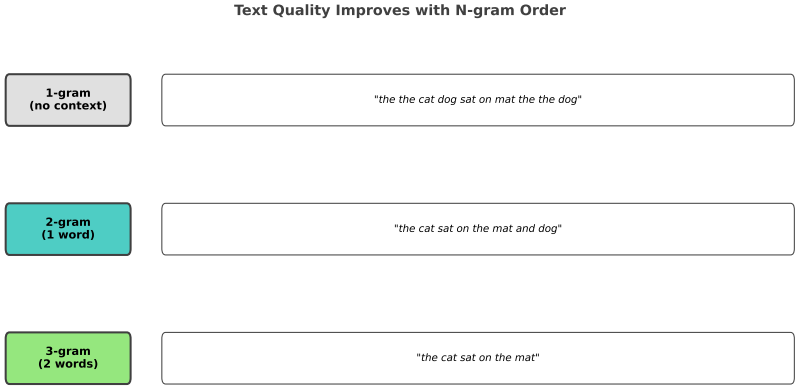
Week 1 - From Dice to Text Prediction

NLP Course 2025

October 26, 2025

BSc Discovery-Based Presentation

Same Model, Different Text Quality



Key Insight: Better models predict next words more accurately

Your phone keyboard uses these models - but how do they work?

A Thought Experiment

Rolling a Die

- Six possible outcomes: 1, 2, 3, 4, 5, 6
- Each has probability: $\frac{1}{6} = 0.167$
- No memory: Previous rolls don't matter
- Fair die: All outcomes equally likely

Question: Can we predict the next roll?

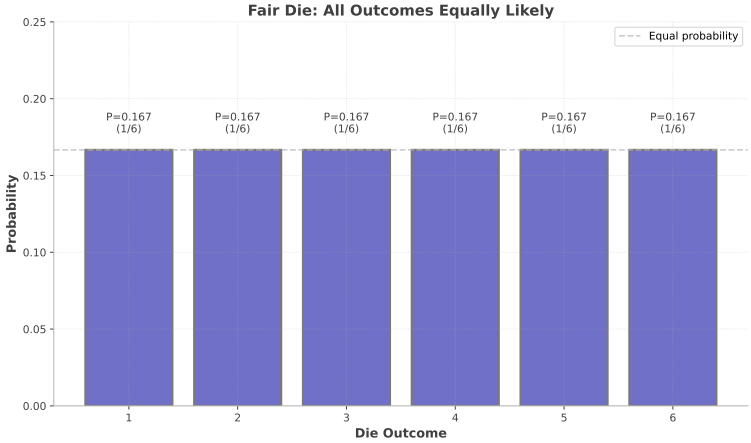
Predicting Words

- Thousands of possible words
- Each has *different* probability
- **Memory matters**: Previous words help
- Not equally likely: "the" more common than "xylophone"

Question: Can we predict the next word?

Both are probability problems - but text prediction is harder

Dice Probability: The Foundation



Key Insight: When all outcomes equally likely, $P(\textit{outcome}) = \frac{1}{\text{number of outcomes}}$

This is the simplest case - text is more complex

From Dice to Text Prediction

What We Know from Dice:

- Probability quantifies uncertainty
- Sum of all probabilities = 1
- More data → better estimates

New Challenges for Text:

- Outcomes NOT equally likely
- Context matters ("the cat" vs "the xylophone")
- How to estimate probabilities?

Our Goal:
Predict next word given previous words

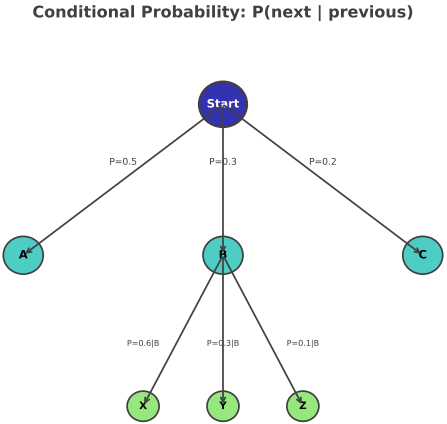
Example:

- Given: "The cat sat on the"
- Predict: "mat" (likely) or "xylophone" (unlikely)

How?
Use **conditional probability** based on what we've seen before

Next: Build conditional probability from first principles

Conditional Probability: A Simple Example



Conditional Probability: The Mathematics

Definition:

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

Reads: "Probability of A given B"

Worked Example with Cards:
Given: Drew a face card (J, Q, K)
Question: What's probability it's a King?

- Total face cards: 12 (4 Jacks, 4 Queens, 4 Kings)
- Kings among face cards: 4
- $P(\text{King}|\text{Face}) = \frac{4}{12} = \frac{1}{3}$

For Text Prediction:

$$P(w_n|w_1, w_2, ..., w_{n-1})$$

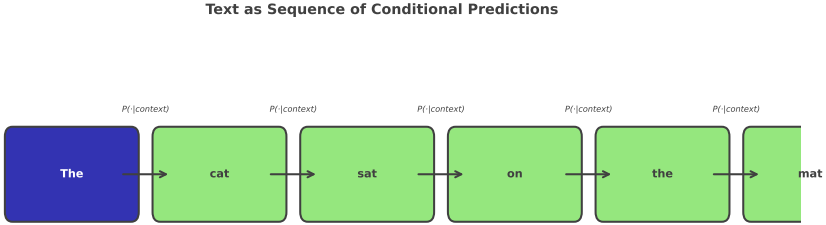
Reads: "Probability of word n given all previous words"

Example:
Given: "The cat sat on the"
Question: What's $P(\text{mat}|\text{the cat sat on the})$?

Answer: Count how many times we've seen this pattern!

Conditional probability is the foundation of language modeling

Text as a Sequence of Decisions



Key Insight: Each word is a prediction given all previous words

But how do we get these probabilities?

Word Prediction: The Core Problem

The Chain Rule:
For a sequence $w_1, w_2, ..., w_n$:

$$\begin{aligned} P(w_1, w_2, ..., w_n) = & P(w_1) \\ & \times P(w_2|w_1) \\ & \times P(w_3|w_1, w_2) \\ & \times ... \\ & \times P(w_n|w_1, ..., w_{n-1}) \end{aligned}$$

Challenge: How to estimate $P(w_n|w_1, ..., w_{n-1})$?

Example: "The cat sat":

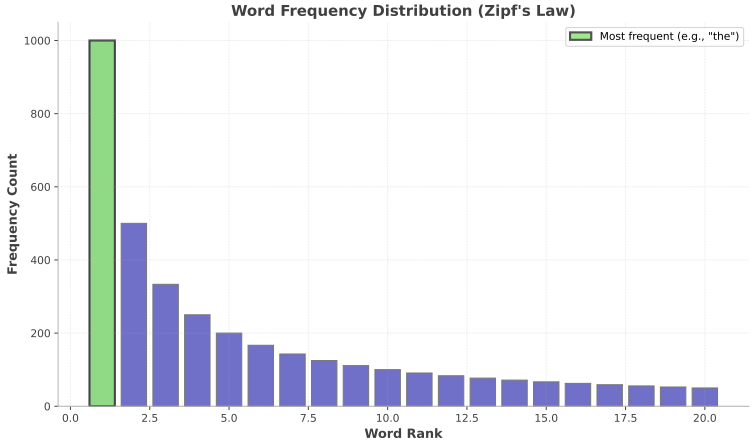
$$\begin{aligned} P(\text{the, cat, sat}) = & P(\text{the}) \\ & \times P(\text{cat}|\text{the}) \\ & \times P(\text{sat}|\text{the, cat}) \end{aligned}$$

Problem:
Infinitely many possible histories!

Solution:
Make a simplifying assumption (coming next)

We need a practical way to estimate these probabilities

The Solution: Count What We've Seen



Key Insight: Probability ≈ Frequency in large corpus

This is Maximum Likelihood Estimation

Maximum Likelihood Estimation (MLE)

The Idea:
Estimate probabilities from observed counts

Formula:

$$P(w|context) = \frac{\text{count}(context, w)}{\text{count}(context)}$$

Example from Shakespeare:

- count("to be") = 150 times
- count("to be or") = 42 times
- $P(\text{or}|\text{to be}) = \frac{42}{150} = 0.28$

Why This Works:

- Law of Large Numbers
- As corpus size $\to \infty$, relative frequency $\to$ true probability
- Real corpora: millions/billions of words

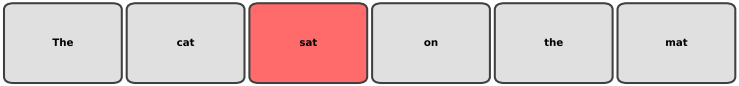
When to Use:

- Have large text corpus
- Want simple, interpretable model
- Need fast training and inference

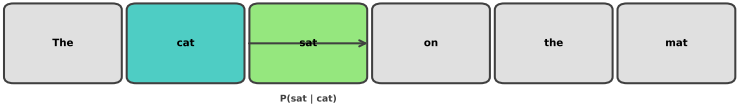
MLE is simple but effective - used in production systems

N-gram Models: The Markov Assumption

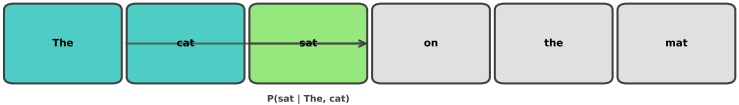
Unigram (n=1): No Context



Bigram (n=2): Previous Word



Trigram (n=3): Two Previous Words



N-gram Models: Making It Practical

The Markov Assumption:

$$P(w_i|w_1, ..., w_{i-1}) \approx P(w_i|w_{i-n+1}, ..., w_{i-1})$$

Only last $n-1$ words matter

Different N-gram Models:

- **Unigram** ($n = 1$): $P(w_i)$
- **Bigram** ($n = 2$): $P(w_i|w_{i-1})$
- **Trigram** ($n = 3$): $P(w_i|w_{i-2}, w_{i-1})$

Why This Helps:

- Reduces number of parameters
- More data per pattern
- Tractable computation

Trade-off:

- Small n: Fast, robust, but misses long-range dependencies
- Large n: Captures context, but sparse data

Typical Choice: $n = 3$ (trigram) balances both

N-grams are the workhorse of statistical language modeling

Worked Example: Bigram Probabilities

Given Shakespeare corpus extract:
"To be or not to be that is the question whether"

Task: Compute $P(\text{be}|\text{to})$

Step 1: Count all bigrams starting with "to"

Bigram	Count
(to, be)	2

Step 2: Count total occurrences of "to"
$\text{count}(\text{to}) = 2$

Step 3: Apply MLE formula

$$P(\text{be}|\text{to}) = \frac{\text{count}(\text{to, be})}{\text{count}(\text{to})} = \frac{2}{2} = 1.0$$

Interpretation: In this tiny corpus, "to" is always followed by "be"!

Real corpora give more nuanced probabilities

Checkpoint: Test Your Understanding

Quick Quiz

Question 1:
Given corpus: "the cat sat on the mat the dog"
What is $P(\text{cat}|\text{the})$?

A) $1/3$
B) $1/2$
C) $2/3$
D) $1/8$

Question 2:
Why do we use the Markov assumption?

A) It's always correct
B) Reduces parameters
C) Improves accuracy
D) Runs faster

Answer 1: **A) 1/3**

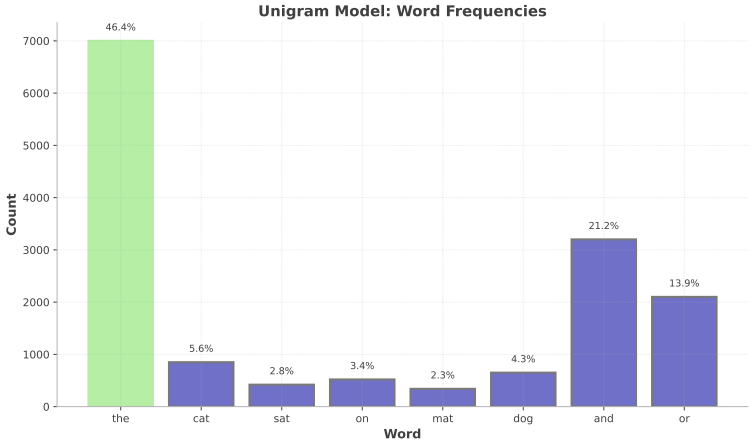
- $\text{count}(\text{the}) = 3$
- $\text{count}(\text{the, cat}) = 1$
- $P(\text{cat}|\text{the}) = 1/3$

Answer 2: **B) Reduces parameters**

- Without it: Infinite histories
- With it: Tractable parameter count
- Trade-off: Lose long-range info

Understanding these foundations is critical for what comes next

Unigram Model: The Simplest Baseline



Key Insight: Each word has fixed probability regardless of context

Simple but ignores all context - like random word sampling

Unigram Model: When Context Doesn't Matter

Formula:

$$P(w_1, w_2, \ldots, w_n) = \prod_{i=1}^{n} P(w_i)$$

Each word independent

MLE Estimation:

$$P(w) = \frac{\text{count}(w)}{\text{total words}}$$

Example:

- Total words: 1,000,000
- count("the") = 70,000
- $P(\text{the}) = 0.07$

When to Use:

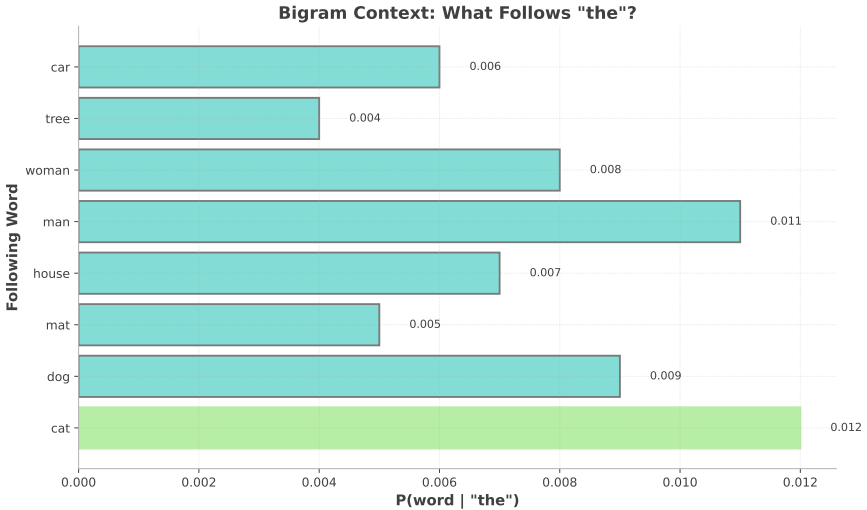
- Baseline comparison
- Document classification
- Bag-of-words features
- When word order truly doesn't matter

Limitations:

- Generates nonsense: "the the the cat dog"
- No grammar
- No meaning
- High perplexity

Unigrams are weak for generation but useful for other NLP tasks

Bigram Model: One Word of Memory



Key Insight: Previous word dramatically narrows possibilities

Bigram Model: The Sweet Spot

Formula:

$$P(w_i|w_{i-1})$$

Condition on previous word only

MLE Estimation:

$$P(w_2|w_1) = \frac{\text{count}(w_1, w_2)}{\text{count}(w_1)}$$

Worked Example:

- count("the") = 70,000
- count("the cat") = 850
- $P(\text{cat}|\text{the}) = \frac{850}{70000} = 0.012$

When to Use:

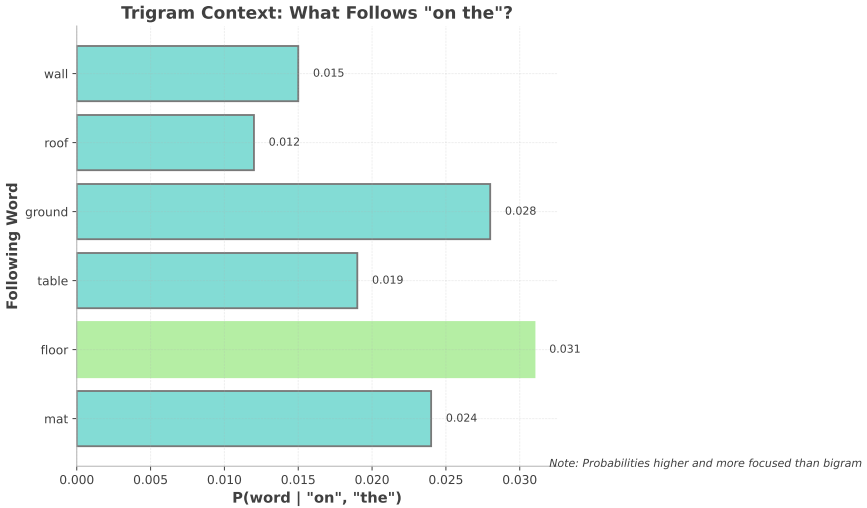
- Real-time applications (autocomplete)
- Limited memory/computation
- Reasonable text quality needed
- Standard baseline for comparison

Performance:

- Captures local grammar
- Generates coherent phrases
- Moderate perplexity (100-150)
- Still misses long-range dependencies

Bigrams are widely used - good balance of simplicity and performance

Trigram Model: Two Words of Memory



Key Insight: Two-word context captures richer patterns

Trigram Model: More Context, Better Predictions

Formula:

$$P(w_i|w_{i-2}, w_{i-1})$$

Condition on two previous words

MLE Estimation:

$$P(w_3|w_1, w_2) = \frac{\text{count}(w_1, w_2, w_3)}{\text{count}(w_1, w_2)}$$

Example:

- count("on the") = 5,200
- count("on the mat") = 127
- $P(\text{mat}|\text{on the}) = \frac{127}{5200} = 0.024$

Comparison to Bigram:

Model	Perplexity	Quality
Bigram	125	Good
Trigram	78	Better

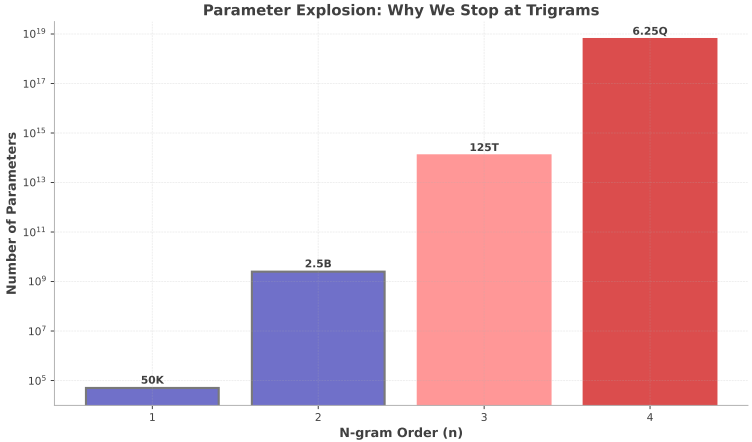
When to Use:

- Sufficient training data
- Quality matters more than speed
- Speech recognition, translation

Limitation: Data sparsity increases

Trigrams are standard in production systems when data permits

Higher-Order N-grams: The Sparsity Wall



Key Insight: Parameters explode exponentially with n

This is why we rarely go beyond trigrams

The Fundamental Tradeoff

Parameter Count:
For vocabulary size V:

- Unigram: V (e.g., 50,000)
- Bigram: V^2 (2.5 billion)
- Trigram: V^3 (125 trillion!)
- 4-gram: V^4 (6,250,000 trillion!)

Reality: Most combinations never seen

The Dilemma:

- More context $\rightarrow$ Better predictions
- More context $\rightarrow$ More parameters
- More parameters $\rightarrow$ Sparse data
- Sparse data $\rightarrow$ Poor estimates

Practical Choice:

- $n = 2$ (bigram): Fast, robust
- $n = 3$ (trigram): Standard
- $n \geq 4$: Rarely used

This limitation motivates smoothing (coming next) and neural models (Week 2)

The Sparsity Problem: Quantified

Experiment: Train on 1 million words, test on held-out data

Model	Seen in Training	Unseen in Test	OOV Rate
Unigram	45,000 words	2,300 words	5%
Bigram	523,000 pairs	87,000 pairs	14%
Trigram	891,000 triples	203,000 triples	19%
4-gram	958,000 quads	347,000 quads	27%

Pattern: As n increases, more unseen combinations

Problem: MLE assigns $P = 0$ to unseen n-grams

Consequence: If any n-gram has $P = 0$, entire sentence gets $P = 0$!

We need a way to handle unseen n-grams without destroying sentence probabilities

Why Zero Probability is Catastrophic

Example Sentence: "The cat sat on the xylophone"

Using Bigram Model:

$$P(\text{sentence}) = P(\text{the}) \times P(\text{cat}|\text{the}) \times P(\text{sat}|\text{cat}) \\ \times P(\text{on}|\text{sat}) \times P(\text{the}|\text{on}) \times P(\text{xylophone}|\text{the})$$

Problem:
If "the xylophone" never appeared in training:

$$P(\text{xylophone}|\text{the}) = \frac{0}{70000} = 0$$

Therefore: $P(\text{entire sentence}) = 0$

Consequence:

- Can't rank this sentence
- Can't generate it
- Perplexity becomes infinite!

A single unseen n-gram destroys everything - we must fix this

Root Cause: The Sparse Data Curse

What We Have:

- Training corpus: 1M words
- Vocabulary: 50K words
- Possible bigrams: $50K^2 = 2.5$ billion
- Observed bigrams: 500K

Coverage: $\frac{500K}{2.5B} = 0.02\%$

We've seen only 0.02% of possible bigrams!

The Mathematics:

- No smoothing: $P = 0$ for 99.98% of bigrams
- Sentence of length 10: $P(\text{sentence}) = 0$ in 95% of cases
- Useless for real applications

Root Cause:
MLE assumes if we haven't seen it, it's impossible

Reality:
Unseen $\neq$ Impossible

We need to reserve some probability mass for unseen events

The Solution: Smoothing

∞

Laplace Smoothing: The Simplest Fix

The Idea:
Pretend we've seen every n-gram one extra time

Formula (Bigram):

$$P_{smooth}(w_2|w_1) = \frac{\text{count}(w_1, w_2) + 1}{\text{count}(w_1) + V}$$

where V = vocabulary size

Worked Example:

- count("the") = 70,000
- count("the cat") = 850
- V = 50,000

$$P(\text{cat}|\text{the}) = \frac{850 + 1}{70000 + 50000} = \frac{851}{120000} = 0.0071$$

For Unseen N-gram:
count("the xylophone") = 0

$$P(\text{xylophone}|\text{the}) = \frac{0 + 1}{70000 + 50000} = \frac{1}{120000} = 8.3 \times 10^{-6}$$

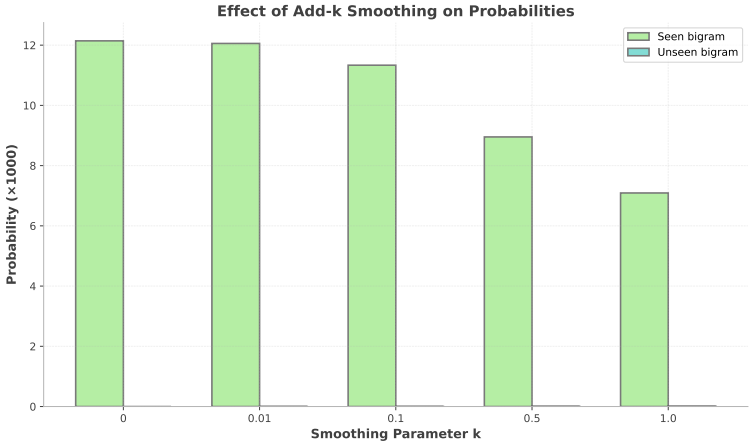
Small but non-zero!

Properties:

- Simple to implement
- Never gives $P = 0$
- Too much mass to rare events
- Hurts frequent events

Laplace smoothing is too aggressive for language - we need something better

Add-k Smoothing: A Better Balance



Key Insight: Add $k < 1$ instead of 1 to balance probability redistribution

Typical values: $k = 0.01$ to $k = 0.5$

Advanced Smoothing: Kneser-Ney

The Insight:
Not all words are equally likely in new contexts

Example:

- "Francisco" appears often
- But only after "San"
- Shouldn't get high probability after other words!

Solution:
Count *how many different contexts* a word appears in, not just total frequency

Kneser-Ney Benefits:

- State-of-the-art for n-grams
- 15-20% perplexity improvement
- Used in production systems

Complexity:

- More sophisticated than add-k
- Requires continuation counts
- Backoff to shorter n-grams

When to Use:
When quality matters most

Full Kneser-Ney derivation beyond BSc scope - but know it exists

Validation: Smoothing Improves Performance

Experimental Setup:
Train on 1M words, test on 100K held-out words, trigram model

Smoothing Method	Perplexity	Unseen N-gram Handling
None (MLE)	∞	Fails
Add-1 (Laplace)	245	Poor
Add-0.1	156	Good
Add-0.01	132	Good
Kneser-Ney	98	Best

Pattern:

- Smaller k better for large corpora
- Kneser-Ney best overall
- 35% improvement over add-1

Takeaway: Smoothing is not optional - it's essential

Modern systems use sophisticated smoothing as standard

Implementation: Add-k Smoothing

```python
class SmoothBigram:
    def __init__(self, k=0.01):
        self.k = k
        self.counts = defaultdict(
            lambda: defaultdict(int)
        )
        self.vocab = set()

    def train(self, text):
        words = text.split()
        for i in range(len(words)-1):
            w1, w2 = words[i], words[i+1]
            self.counts[w1][w2] += 1
            self.vocab.update([w1, w2])

    def probability(self, w1, w2):
        V = len(self.vocab)
        numerator = self.counts[w1][w2] + self.k
        denominator = sum(
            self.counts[w1].values()
        ) + self.k * V
        return numerator / denominator
```

Key Changes from MLE:

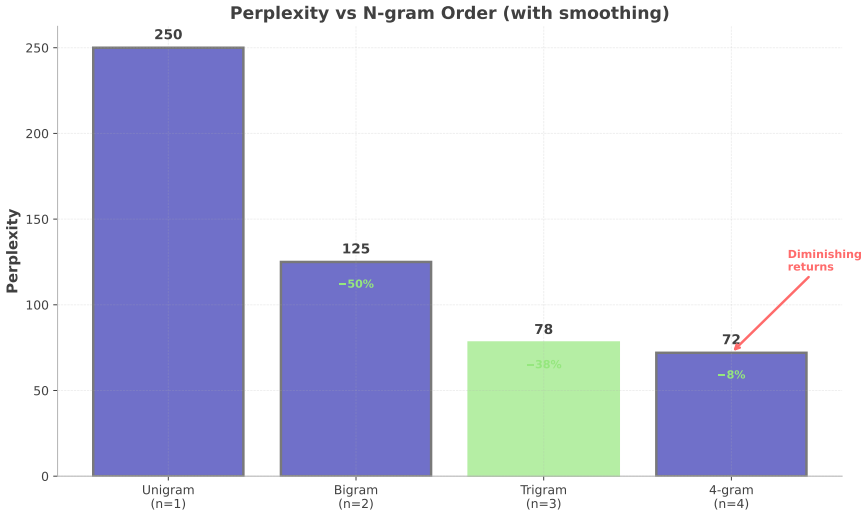
- Add k to numerator
- Add $k \times V$ to denominator
- Never returns 0!

Usage:

```python
model = SmoothBigram(k=0.01)
model.train(corpus)
p = model.probability(
    "the", "xylophone"
)
# Returns small non-zero value
```

Just 4 lines different from MLE - huge impact on performance

Evaluation: How Good is Our Model?



Key Insight: Lower perplexity = better predictions

Perplexity: Measuring Uncertainty

Formula:

$$PP(W) = P(w_1, w_2, ..., w_N)^{-\frac{1}{N}}$$

or equivalently:

$$PP(W) = \sqrt[N]{\frac{1}{P(w_1, ..., w_N)}}$$

Logarithmic Form:

$$\log_2 PP(W) = -\frac{1}{N} \sum_{i=1}^{N} \log_2 P(w_i|context)$$

Interpretation:
"On average, how many equally likely words could come next?"

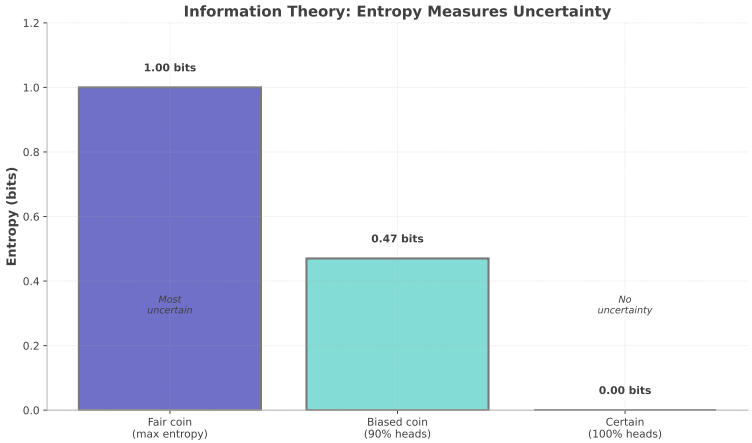
- PP = 100: Choosing from 100 words
- PP = 50: Model is twice as confident
- PP = 10: Very good model
- PP = 1000: Poor model

Properties:

- Lower is better
- Minimized by true distribution
- Inverse of geometric mean probability

Perplexity connects probability to human intuition about uncertainty

Information Theory: The Foundations



Key Insight: Rare events carry more information than common events

Claude Shannon founded information theory in 1948

Shannon's Information Theory

Entropy (Uncertainty):

$$H(X) = -\sum_{x} P(x) \log_2 P(x)$$

Measures average information per symbol

Cross-Entropy:

$$H(P, Q) = -\sum_{x} P(x) \log_2 Q(x)$$

Measures information loss when using Q to approximate P

Relationship to Perplexity:

$$PP = 2^{H(P,Q)}$$

Example - Coin Flip:
Fair coin: $P(\text{heads}) = 0.5$

$$H = -0.5 \log_2(0.5) - 0.5 \log_2(0.5) = 1 \text{ bit}$$

Example - English Text:
Entropy $\approx$ 1-2 bits per character

Why This Matters:

- Fundamental limits on compression
- Optimal encoding
- Neural models minimize cross-entropy

Information theory is the mathematical foundation of all NLP

Worked Example: Computing Perplexity

Given: Bigram model, test sentence: "the cat sat"

Model Probabilities:

- $P(\text{the}) = 0.07$
- $P(\text{cat}|\text{the}) = 0.012$
- $P(\text{sat}|\text{cat}) = 0.08$

Step 1: Compute sentence probability

$$\begin{aligned} P(\text{the, cat, sat}) &= P(\text{the}) \times P(\text{cat}|\text{the}) \times P(\text{sat}|\text{cat}) \\ &= 0.07 \times 0.012 \times 0.08 \\ &= 0.0000672 \end{aligned}$$

Step 2: Apply perplexity formula

$$PP = (0.0000672)^{-\frac{1}{3}} = (0.0000672)^{-0.333} = 126.7$$

Interpretation: Model is as uncertain as picking from 127 equally likely words

Lower perplexity means higher confidence in predictions

When NOT to Use N-grams

Failure Cases:

- **Long-range dependencies**:
  "The author, who wrote several books about..., was awarded a prize."
  ("author" $\rightarrow$ "was", but 10+ words apart)
- **Semantic understanding**:
  "The bank is closed" vs "The river bank"
  (Same n-grams, different meanings)
- **Rare but valid constructions**:
  Creative language, poetry, technical jargon

Better Alternatives:

- **Neural language models** (Week 2):
  Learn distributed representations
- **RNN/LSTM** (Week 3):
  Unbounded context window
- **Transformers** (Week 5):
  Direct long-range connections

When to Stick with N-grams:

- Speed critical
- Interpretability needed
- Limited data
- Baseline comparison

Know your model's limitations - choose the right tool for the job

Common Pitfalls and How to Avoid Them

Pitfall 1: Choosing n

- Too small: Misses context
- Too large: Data sparsity
- Solution: Start with trigrams, validate on held-out data

Pitfall 2: Forgetting smoothing

- MLE gives zero probabilities
- Solution: Always use smoothing in production

Pitfall 3: Train/test contamination

- Testing on training data
- Solution: Strict data splits

Pitfall 4: Vocabulary mismatch

- Test words not in training vocab
- Solution: UNK token for rare words

Pitfall 5: Computational explosion

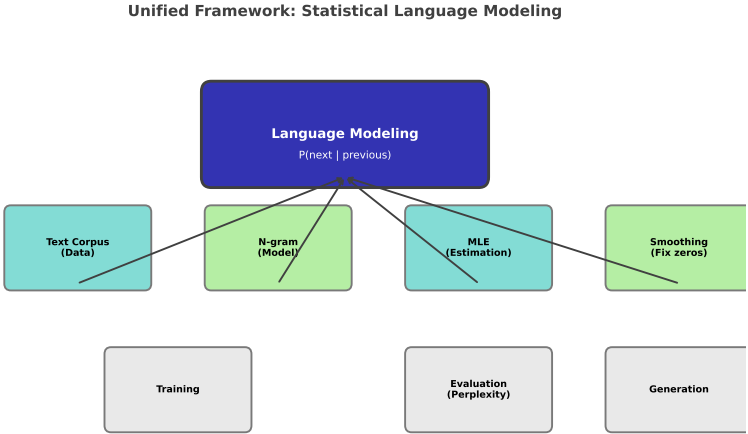
- Storing all n-grams
- Solution: Pruning, hashing tricks

Pitfall 6: Overfitting to corpus

- Model learns corpus-specific patterns
- Solution: Large, diverse training data

Awareness of pitfalls is half the battle - test rigorously

Unified View: Everything is Conditional Probability



Key Takeaways

1. **Conditional probability is the foundation**
   Text prediction is estimating $P(w_n|w_1, ..., w_{n-1})$
2. **N-grams make it practical**
   Markov assumption limits context to last $n-1$ words
3. **MLE estimates from counts**
   $P(w|context) = \frac{\text{count}(context, w)}{\text{count}(context)}$
4. **Smoothing is essential**
   Unseen n-grams get non-zero probability
5. **Perplexity measures quality**
   Lower perplexity = better predictions
6. **Trade-offs everywhere**
   Context vs sparsity, speed vs quality, simplicity vs sophistication

Master these foundations - they underpin all of NLP

Next: Hands-On Practice

Lab Notebook Activities:

1. Build unigram model from scratch
2. Implement bigram with MLE
3. Add smoothing (compare methods)
4. Generate text and compare quality
5. Compute perplexity on test data
6. Visualize n-gram distributions
7. Experiment with different n values

What You'll Learn:

- Hands-on probability estimation
- See smoothing's impact
- Debug zero probability issues
- Understand perplexity intuitively
- Reproduce presentation charts

Corpus: Shakespeare sonnets (interesting patterns!)

Let's build some language models!

The lab cements understanding - theory alone is not enough