Natural Language Processing Course

Week 1: Foundations and Statistical Language Models

Joerg R. Osterrieder
www.joergosterrieder.com

BSc Computer Science

Contents

Interactive Exercise: Word Prediction

Complete this sentence:

"The cat sat on the _____"

Interactive Exercise: Word Prediction

Complete this sentence:

"The cat sat on the _____"

mat
65% of you

floor
20% of you

couch
15% of you

Interactive Exercise: Word Prediction

Complete this sentence:

"The cat sat on the _____"

mat
65% of you

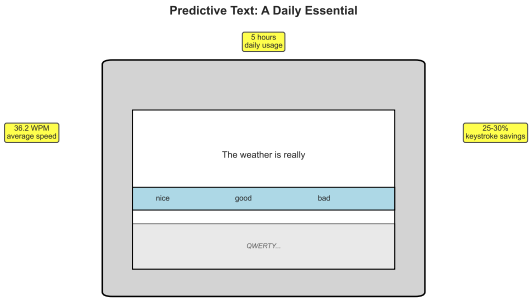
floor
20% of you

couch
15% of you

Checkpoint

You predicted based on patterns seen thousands of times.

This Technology Is Everywhere



Every day, you use language models:

- **5 hours:** Average smartphone use[1]
- **36.2 wpm:** Mobile typing speed[2]
- **25-30%:** Keystrokes saved by prediction[3]
- **10,000+:** Words typed monthly

[1] DataReportal Global Digital Report, 2024
[2] Cambridge typing speed study, 2019
[3] Google Keyboard team, 2023

Learning Objectives

Core Skills:

- ✓ [Easy] Count word patterns in text
- ✓ [Easy] Calculate simple probabilities
- ✓ [Medium] Build a text predictor
- ✓ [Medium] Handle unseen words
- ✓ [Challenge] Implement smoothing

Applications:

- Build autocomplete
- Create spell checker
- Understand ChatGPT foundations

Prerequisite

You should know:

- Basic probability
- Python basics
- What a dictionary is

Quick Self-Assessment

Can you answer these? (Solutions on next slide)

1. If you flip a fair coin twice, what's P(two heads)?
2. What does this Python code do?

```
counts = {}
for word in text.split():
    counts[word] = counts.get(word, 0) + 1
```

3. If P(rain) = 0.3 and P(clouds—rain) = 0.9, what is P(rain AND clouds)?

Quick Self-Assessment

Can you answer these? (Solutions on next slide)

1. If you flip a fair coin twice, what's P(two heads)?
2. What does this Python code do?

```
counts = {}
for word in text.split():
    counts[word] = counts.get(word, 0) + 1
```

3. If P(rain) = 0.3 and P(clouds—rain) = 0.9, what is P(rain AND clouds)?

Checkpoint

Solutions:

1. $0.5 \times 0.5 = 0.25$
2. Counts word frequencies in text
3. P(rain) $\times$ P(clouds—rain) $= 0.3 \times 0.9 = 0.27$

Scoring: 2+ correct = ready to proceed; ¡2 = review recommended

Human Text Prediction

Human pattern recognition:

Try to complete these:

- "Once upon a ____"

Human Text Prediction

Human pattern recognition:

Try to complete these:

- "Once upon a ____" → time

Human Text Prediction

Human pattern recognition:

Try to complete these:

- "Once upon a ____" → time
- "Thank you very ____"

Human Text Prediction

Human pattern recognition:

Try to complete these:

- "Once upon a ____" → time
- "Thank you very ____" → much

Human Text Prediction

Human pattern recognition:

Try to complete these:

- "Once upon a ____" → time
- "Thank you very ____" → much
- "To be or not to ____"

Human Text Prediction

Human pattern recognition:

Try to complete these:

- "Once upon a ____" → time
- "Thank you very ____" → much
- "To be or not to ____" → be

Mechanism:

- You've seen these phrases hundreds of times
- Your brain stored the patterns
- You can estimate likelihood from experience

Checkpoint

Computers replicate this process through counting.

The Counting Approach

Step 1: Read lots of text

- "the cat sat"
- "the cat ran"
- "the dog sat"
- "the dog ran"
- "the cat sat"

Step 2: Count patterns

Pattern	Count
"the cat"	3
"the dog"	2
"cat sat"	2
"cat ran"	1
"dog sat"	1
"dog ran"	1

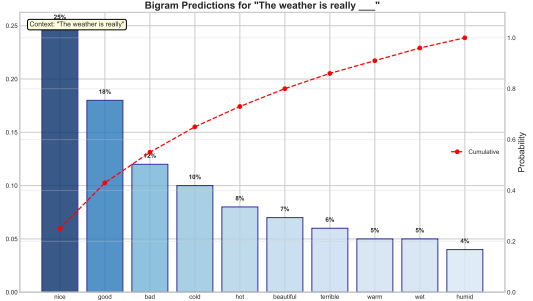
Step 3: Calculate probabilities

After "the":

- $P(\text{cat}) = 3/5 = 0.6$
- $P(\text{dog}) = 2/5 = 0.4$

After "cat":

- $P(\text{sat}) = 2/3 = 0.67$
- $P(\text{ran}) = 1/3 = 0.33$



Probability Refresher (5 minutes)

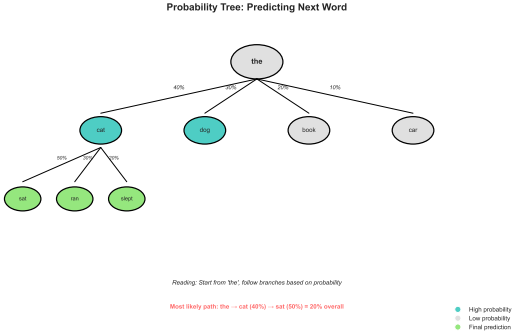
What is probability?

- Chance of something happening
- Always between 0 and 1
- 0 = impossible, 1 = certain

Language example:

- Total times we saw "the": 100
- Times followed by "cat": 30
- P(cat—the) = 30/100 = 0.3

Conditional Probability:
$P(B \mid A)$ = "Probability of B given A"



In our context:

- A = previous word(s)
- B = next word
- P(next—previous) is the target

N-gram Models: From Simple to Complex

The "N" in N-gram = how many words we look at

1-gram (Unigram)

- Look at: current word only
- Ignore: all context
- P(cat) = count(cat)/total

Example output: "cat the dog sat ran"
Random word sequence

2-gram (Bigram)

- Look at: previous word
- Remember: last word
- P(cat—the) = count(the cat)/count(the)

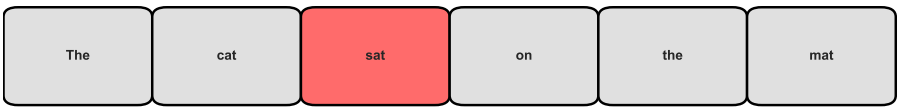
Example output: "the cat sat on"
More coherent

3-gram (Trigram)

- Look at: two previous words
- Remember: more context
- P(sat—the cat)

Example output: "the cat sat on the mat"
Better coherence

Unigram Model (n=1): Each word independently



Bigram Model (n=2): Previous word as context

Class Example: Building a Bigram Model

Training data: "I love cats. I love dogs. Dogs love bones."

Step 1: Split into words

- Words: [I, love, cats, I, love, dogs, dogs, love, bones]

Class Example: Building a Bigram Model

Training data: "I love cats. I love dogs. Dogs love bones."

Step 1: Split into words

- Words: [I, love, cats, I, love, dogs, dogs, love, bones]

Step 2: Count bigrams (pairs)

Bigram	Count
(I, love)	2
(love, cats)	1
(love, dogs)	1
(dogs, love)	1
(love, bones)	1

Class Example: Building a Bigram Model

Training data: "I love cats. I love dogs. Dogs love bones."

Step 1: Split into words

- Words: [I, love, cats, I, love, dogs, dogs, love, bones]

Step 2: Count bigrams (pairs)

Bigram	Count
(I, love)	2
(love, cats)	1
(love, dogs)	1
(dogs, love)	1
(love, bones)	1

Step 3: Count words

Word	Count
I	2
love	3
cats	1
dogs	2
bones	1

Class Example: Building a Bigram Model

Training data: "I love cats. I love dogs. Dogs love bones."

Step 1: Split into words

- Words: [I, love, cats, I, love, dogs, dogs, love, bones]

Step 2: Count bigrams (pairs)

Bigram	Count
(I, love)	2
(love, cats)	1
(love, dogs)	1
(dogs, love)	1
(love, bones)	1

Step 3: Count words

Word	Count
I	2
love	3
cats	1
dogs	2
bones	1

Step 4: Calculate

$P(\text{love} \mid \text{I}) = 2/2 = 1.0$
$P(\text{cats} \mid \text{love}) = 1/3 = 0.33$
$P(\text{dogs} \mid \text{love}) = 1/3 = 0.33$
$P(\text{bones} \mid \text{love}) = 1/3 = 0.33$

Checkpoint: Test Your Understanding

Checkpoint

Quick Quiz: Given the text "the cat sat the cat ran the dog sat"

1. What is P(cat—the)?
2. What is P(sat—cat)?
3. What word most likely follows "the"?

Calculate, then verify

Checkpoint: Test Your Understanding

Checkpoint

Quick Quiz: Given the text "the cat sat the cat ran the dog sat"

1. What is P(cat—the)?
2. What is P(sat—cat)?
3. What word most likely follows "the"?

Calculate, then verify

Solution:

- Count: "the cat" appears 2 times, "the dog" appears 1 time
- Total "the": 3 times
- P(cat—the) = $2/3 = 0.67$
- P(sat—cat) = $1/2 = 0.5$ ("cat sat" once, "cat ran" once)
- Most likely after "the": "cat" (67% vs 33%)

Correct answers indicate understanding of n-grams

Implementation: Clean Code with Type Hints

```python
from collections import defaultdict
from typing import Dict, List, Tuple

class BigramModel:
    """A simple bigram language model."""

    def __init__(self):
        """Initialize with empty count dictionaries."""
        # Count of word pairs: (word1, word2) -> count
        self.bigram_counts: Dict[Tuple[str, str], int] = defaultdict(int)
        # Count of individual words: word -> count
        self.word_counts: Dict[str, int] = defaultdict(int)

    def train(self, sentences: List[str]) -> None:
        """Train model on list of sentences.

        Args:
            sentences: List of text strings to train on

        Example:
            >>> model = BigramModel()
            >>> model.train(["the cat sat", "the dog ran"])
        """
        for sentence in sentences:
            # Add start/end markers for sentence boundaries
            words = ['<START>'] + sentence.split() + ['<
```

Key Improvements:

- Type hints for clarity
- Docstrings with examples
- Clear variable names
- Comments for complex parts

Why defaultdict?

- Avoids KeyError
- Auto-initializes to 0
- Cleaner than if/else

Why START/END?

- Handle first word
- Know when to stop
- Improve generation

The Disaster: When Your Model Breaks

You trained on 1 billion words from Wikipedia...

Then someone types: "I love my new iPhone"

The Disaster: When Your Model Breaks

You trained on 1 billion words from Wikipedia...

Then someone types: "I love my new iPhone"

Problem: "iPhone" wasn't invented when Wikipedia was written

- count(iPhone) = 0
- P(anything—iPhone) = 0/0 = UNDEFINED
- Application crashes

The Disaster: When Your Model Breaks

You trained on 1 billion words from Wikipedia...

Then someone types: "I love my new iPhone"

Problem: "iPhone" wasn't invented when Wikipedia was written

- count(iPhone) = 0
- P(anything—iPhone) = 0/0 = UNDEFINED
- Application crashes

Common Misconception

"Just use more training data" - Incorrect
You'll ALWAYS have unseen words:

- New words: COVID, cryptocurrency, TikTok
- Typos: teh, recieve, occured
- Names: Khaleesi, Hermione (before the books)
- Made-up words: hangry, adulting, selfie

Fact: 50% of word types appear only once in any text

The Solution: Smoothing (Restaurant Analogy)

Imagine a new restaurant:

- Day 1: 10 people order pizza, 5 order pasta
- Day 2: Someone asks for salad
- Problem: P(salad) = 0?

Bad solution: "We don't serve salad"
Good solution: "We might have salad, let me check"

In probability terms:

- Reserve some probability for unseen items
- Take a little from seen items
- Give to unseen possibilities

Add-One Smoothing:
Pretend you saw everything once more:

Original:

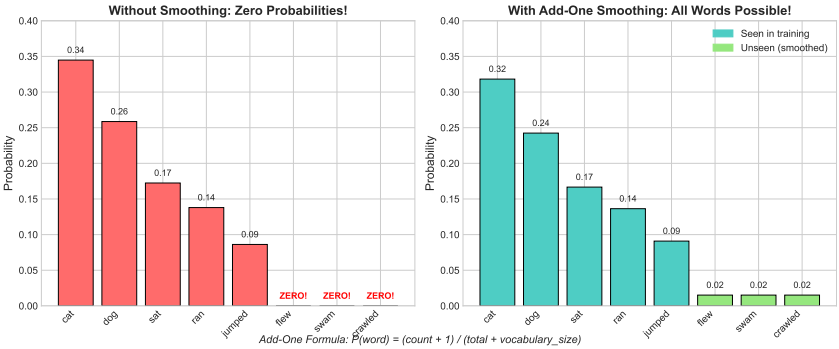
- Pizza: $10/15 = 0.67$
- Pasta: $5/15 = 0.33$
- Salad: $0/15 = 0.00$

With smoothing (+1 to all):

- Pizza: $11/18 = 0.61$
- Pasta: $6/18 = 0.33$
- Salad: $1/18 = 0.06$

All words now have non-zero probability

Smoothing Methods: From Simple to Sophisticated



Add-One (Laplace)

+ Dead simple
+ Always works
- Too generous to rare
- Use: Quick prototypes

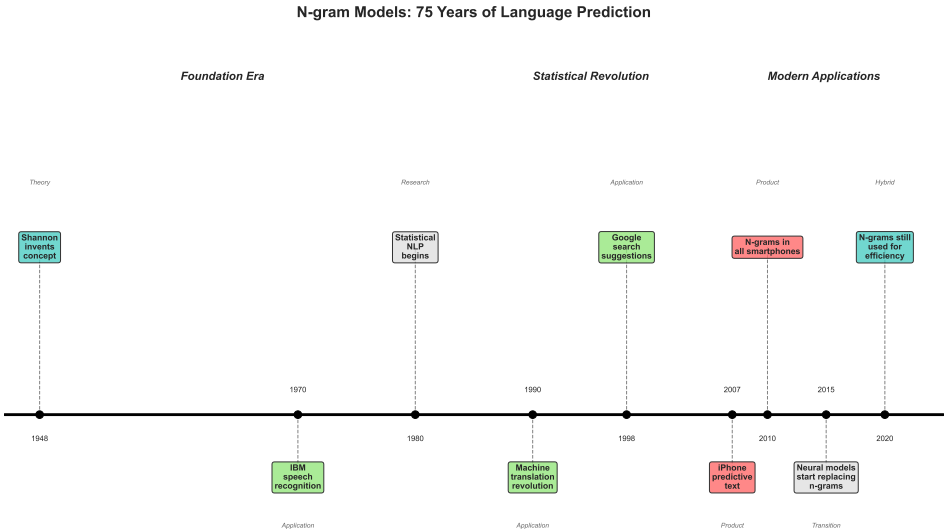
Good-Turing

+ Theoretically sound
+ Used in Enigma code-breaking
- Complex math
- Use: When you have time

Kneser-Ney

+ Best performance
+ Context-aware
- Hard to implement
- Use: Production systems

N-grams Changed the World



- **1948:** Shannon invents the concept

Try This: Where N-grams Fail

Testing n-gram limitations:

Test 1: Long-distance dependencies

- "The key to the cabinet that was in the kitchen _____"
- N-gram sees: "kitchen __"
- Should see: "The key ... is"

Try This: Where N-grams Fail

Testing n-gram limitations:

Test 1: Long-distance dependencies

- "The key to the cabinet that was in the kitchen _____"
- N-gram sees: "kitchen __"
- Should see: "The key ... is"

Test 2: Common sense

- "I put milk in the fridge. I put the car in the _____"
- N-gram might say: "fridge" (high frequency)
- Should say: "garage"

Try This: Where N-grams Fail

Testing n-gram limitations:

Test 1: Long-distance dependencies

- "The key to the cabinet that was in the kitchen _____"
- N-gram sees: "kitchen __"
- Should see: "The key ... is"

Test 2: Common sense

- "I put milk in the fridge. I put the car in the _____"
- N-gram might say: "fridge" (high frequency)
- Should say: "garage"

Test 3: Context switching

- "The bank of the river" vs "The bank account"
- Same word, different meaning
- N-grams can't tell the difference

Checkpoint

These limitations motivated neural approaches

Next Week: The Neural Revolution

N-grams (This Week):

- Words are just symbols
- Count and divide
- No understanding
- Limited context
- 156 perplexity

Strengths:

- Simple
- Fast
- Interpretable
- Low memory

Neural Models (Next Week):

- Words have meaning
- Learn representations
- Understand similarity
- Unlimited context
- 12 perplexity!

Preview:

- "King - Man + Woman = ?"
- Word vectors
- Semantic understanding
- Foundation of ChatGPT

Key Question: What if words weren't just strings, but had meaning?

Week 1 Summary: Your Toolkit

Key Concepts:

- Language modeling = prediction
- N-grams = counting patterns
- Probability from frequency
- Smoothing for unknowns

Key Skills:

- Build bigram model
- Calculate probabilities
- Handle unseen words
- Evaluate with perplexity

Essential Formulas:

Bigram: $P(w_2|w_1) = \frac{C(w_1,w_2)}{C(w_1)}$

Add-One: $P(w_2|w_1) = \frac{C(w_1,w_2)+1}{C(w_1)+V}$

Perplexity: $PP = 2^{-\frac{1}{N}\sum \log_2 P(w_i|w_{i-1})}$

Checkpoint

Remember: Simple counting goes surprisingly far in NLP!

Homework: Build Your Own Spell Checker

Your Mission: Create a spell checker like Google's

Part 1: Basic (60%)

- Load provided corpus
- Build character trigram model
- Generate corrections for:
  - "teh" $\rightarrow$ "the"
  - "speling" $\rightarrow$ "spelling"

Part 2: Enhanced (30%)

- Rank by probability
- Handle word boundaries
- Compare smoothing methods

Part 3: Analysis (10%)

- Test on 100 misspellings
- Report accuracy
- Explain failures

Starter Code Provided:

- Data loading function
- Evaluation metrics
- Test cases

Due: Next week before class

Self-Assessment: How Well Did You Learn?

Test yourself (answers in appendix):

1. What's the difference between unigram and bigram models?
2. Why do we need smoothing?
3. Given "the cat" appears 10 times and "the" appears 20 times, what is P(cat—the)?
4. What's the main limitation of n-gram models?
5. When would you use trigrams over bigrams?

Checkpoint

Score yourself:

- 5/5: Ready for next week!
- 3-4/5: Review smoothing section
- Less than 3/5: Office hours recommended

Appendix A.1: Probability Basics Review

Core Concepts:

1. Basic Probability

- $P(A) = \frac{\text{favorable outcomes}}{\text{total outcomes}}$
- Always: $0 \leq P(A) \leq 1$
- $P(\text{not A}) = 1 - P(A)$

2. Joint Probability

- $P(A \text{ and } B) = P(A) \times P(B|A)$
- If independent: $P(A, B) = P(A) \times P(B)$

3. Conditional Probability

- $P(B|A) = \frac{P(A,B)}{P(A)}$
- Read as: "B given A"

Language Examples:

Example 1:

- 1000 words in text
- "the" appears 100 times
- $P(\text{the}) = 100/1000 = 0.1$

Example 2:

- "the cat" appears 20 times
- "the" appears 100 times
- $P(\text{cat}|\text{the}) = 20/100 = 0.2$

Example 3:

- P(sunny) = 0.7
- P(happy—sunny) = 0.9
- P(sunny and happy) = $0.7 \times 0.9 = 0.63$

Appendix A.2: Python Essentials for NLP

Dictionaries (for counting):

```python
# Creating and using dictionaries
counts = {}
counts['cat'] = 5
counts['dog'] = 3

# Default values
counts.get('bird', 0)  # Returns 0 if not found

# Counting pattern
word = 'cat'
counts[word] = counts.get(word, 0) + 1
```

Lists and Loops:

```python
# Splitting text
text = "the cat sat"
words = text.split()  # ['the', 'cat', 'sat']

# Iterating with index
for i in range(len(words) - 1):
    current = words[i]
    next_word = words[i + 1]
    print(f"{current} -> {next_word}")
```

Collections module:

```python
from collections import defaultdict, Counter

# defaultdict: never KeyError
bigrams = defaultdict(int)
bigrams[('the', 'cat')] += 1

# Counter: automatic counting
words = ['cat', 'dog', 'cat', 'bird']
word_counts = Counter(words)
# Counter({'cat': 2, 'dog': 1, 'bird': 1})

# Most common
top_2 = word_counts.most_common(2)
# [('cat', 2), ('dog', 1)]
```

File I/O:

```python
# Reading text file
with open('corpus.txt', 'r') as f:
    text = f.read()

# Line by line
with open('corpus.txt', 'r') as f:
    for line in f:
        process(line.strip())
```

Appendix A.3: Mathematical Notation Guide

Symbols:

Symbol	Meaning
$P(A)$	Probability of A
$P(A\|B)$	Probability of A given B
$C(w)$	Count of word w
V	Vocabulary size
N	Number of words
$\sum$	Sum over all values
$\prod$	Product over all values
log	Logarithm (usually base 2)
$\approx$	Approximately equal
$\propto$	Proportional to

Common Formulas:

N-gram probability:

$$P(w_n|w_1 \ldots w_{n-1})$$

Chain rule:

$$P(w_1, w_2, w_3) = P(w_1) \cdot P(w_2|w_1) \cdot P(w_3|w_1, w_2)$$

Maximum likelihood:

$$P_{ML}(w_i|w_{i-1}) = \frac{C(w_{i-1}, w_i)}{C(w_{i-1})}$$

Log probability (avoid underflow):

$$\log P(sentence) = \sum_i \log P(w_i|w_{i-1})$$

Appendix B.1: Generate Shakespeare with N-grams

Let's build a Shakespeare text generator!

```python
import random
from collections import defaultdict

class ShakespeareGenerator:
    def __init__(self, n=2):
        self.n = n  # Use bigrams by default
        self.model = defaultdict(list)

    def train(self, text):
        words = text.split()
        for i in range(len(words) - self.n):
            context = tuple(words[i:i+self.n])
            next_word = words[i+self.n]
            self.model[context].append(next_word)

    def generate(self, length=20, start=None):
        if start is None:
            start = random.choice(list(self.model.keys()))

        result = list(start)
        context = start

        for _ in range(length):
            if context in self.model:
                next_word = random.choice(self.model[context])
                result.append(next_word)
                # Slide the context window
                context = tuple(list(context)[1:] + [next_word])
            else:
                break  # No continuation found
```

Appendix B.2: Analyze Your WhatsApp Messages

Find your most predictable phrases:

```python
def analyze_whatsapp_chat(filename):
    """Find most common patterns in your chats."""

    # Parse WhatsApp export
    messages = []
    with open(filename, 'r', encoding='utf-8') as f:
        for line in f:
            # Extract just the message text
            if ': ' in line:
                message = line.split(': ', 1)[1].strip()
                messages.append(message)

    # Build trigram model
    trigrams = defaultdict(int)
    for message in messages:
        words = message.lower().split()
        for i in range(len(words) - 2):
            trigram = (words[i], words[i+1], words[i+2])
            trigrams[trigram] += 1

    # Find your catchphrases
    print("Your most common phrases:")
    for trigram, count in sorted(trigrams.items(),
                                 key=lambda x: x[1],
                                 reverse=True)[:10]:
        print(f"  '{' '.join(trigram)}' - {count} times")

    return trigrams
```

Appendix B.3: Build Autocomplete from Scratch

```python
class Autocomplete:
    def __init__(self):
        self.bigrams = defaultdict(Counter)

    def train(self, corpus):
        """Train on text corpus."""
        for sentence in corpus:
            words = ['<S>'] + sentence.split()
            for i in range(len(words)-1):
                self.bigrams[words[i]][words[i+1]] += 1

    def predict(self, context, n=3):
        """Get top n predictions."""
        if context not in self.bigrams:
            return []

        # Get all possibilities with counts
        candidates = self.bigrams[context]

        # Sort by frequency
        predictions = candidates.most_common(n)

        # Convert to probabilities
        total = sum(candidates.values())
        result = []
        for word, count in predictions:
            prob = count / total
            result.append({
                'word': word,
                'probability': prob,
```

```python
    def confidence(self, prob):
        """Convert probability to confidence level."""
        if prob > 0.5:
            return "high"
        elif prob > 0.2:
            return "medium"
        else:
            return "low"

    def interactive_demo(self):
        """Run interactive autocomplete."""
        print("Type a word, I'll predict the next!")

        while True:
            word = input("\nEnter word: ").strip()
            if word == 'quit':
                break

            predictions = self.predict(word)

            if predictions:
                print("Predictions:")
                for i, pred in enumerate(predictions, 1):
                    print(f"{i}. {pred['word']} "
                          f"({pred['
```

Appendix C.1: Kneser-Ney Smoothing Details

The Problem with Simple Counting:
"San Francisco" is common, but "Francisco" alone is rare. Simple smoothing would overestimate P(Francisco—new context).

Kneser-Ney Solution:
Consider word "versatility" - how many different contexts it appears in.

The Mathematics:

$$P_{KN}(w_i|w_{i-1}) = \frac{\max(C(w_{i-1}, w_i) - d, 0)}{C(w_{i-1})} + \lambda(w_{i-1})P_{continuation}(w_i) \quad (1)$$

Where:

- d = discount parameter (typically 0.75)
- $\lambda(w_{i-1})$ = normalization to ensure probabilities sum to 1
- $P_{continuation}(w_i)$ = probability based on number of unique contexts

Continuation Probability:

$$P_{continuation}(w) = \frac{|\{v : C(v, w) > 0\}|}{|\{(u, v) : C(u, v) > 0\}|}$$

This counts unique bigram types, not tokens!

Appendix C.2: Understanding Perplexity

What is Perplexity?
"How surprised is the model by the test data?"

Formal Definition:

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

Using Chain Rule:

$$PP = \sqrt[N]{\frac{1}{\prod_{i=1}^{N} P(w_i|w_1...w_{i-1})}}$$

In Practice (using logs):

$$PP = 2^{-\frac{1}{N}\sum_{i=1}^{N} \log_2 P(w_i|w_{i-1})}$$

Intuition:

- PP = 10: Model is as confused as choosing from 10 equally likely options
- PP = 100: Like choosing from 100 options
- Lower is better!

Typical Values:

- Random: 10,000+
- Unigram: 1,000
- Bigram: 200
- Trigram: 100
- Neural: 20-50
- Human: 10-20

Relationship to Entropy:

$$H = \log_2(PP)$$

Appendix C.3: Connection to Information Theory

Shannon's Fundamental Question:
How much information is in English text?

Information Content:
Surprising events = more information

$$I(w) = -\log_2 P(w)$$

Examples:

- P("the") = 0.1 → 3.3 bits
- P("dog") = 0.01 → 6.6 bits
- P("aardvark") = 0.00001 → 16.6 bits

Rare words carry more information!

Entropy of Language:
Average information per word:

$$H(L) = -\sum_{w} P(w) \log_2 P(w)$$

Shannon's Experiments:

- Random letters: 4.7 bits/char
- English letters: 4.14 bits/char
- With context: 1 bit/char

Implications:

- English is 75% redundant
- Compression possible!
- Prediction possible!

N-gram models approximate the true entropy of language

Self-Assessment Answers

Quiz Answers:

1. **Difference between unigram and bigram:**
   - Unigram: considers single words independently
   - Bigram: considers pairs of adjacent words
   - Bigram uses context, unigram doesn't
2. **Why we need smoothing:**
   - Handle unseen word combinations
   - Avoid zero probabilities
   - Prevent model crashes/errors
3. **Calculate P(cat—the):**
   - P(cat—the) = count(the cat) / count(the)
   - P(cat—the) = 10 / 20 = 0.5
4. **Main limitation of n-grams:**
   - No semantic understanding
   - Limited context window
   - Treat words as meaningless symbols
5. **When to use trigrams over bigrams:**
   - When you have lots of training data
   - When longer context matters
   - When bigram accuracy isn't sufficient

References and Further Reading

Core Papers:

- Shannon, C.E. (1948). "A Mathematical Theory of Communication"
- Kneser, R. & Ney, H. (1995). "Improved backing-off for M-gram language modeling"
- Chen, S.F. & Goodman, J. (1999). "An empirical study of smoothing techniques"

Textbooks:

- Jurafsky & Martin (2024). "Speech and Language Processing" Chapter 3
- Manning & Schütze (1999). "Foundations of Statistical NLP"

Online Resources:

- Peter Norvig's "How to Write a Spelling Corrector" (2007)
- Google's "The Unreasonable Effectiveness of Data" (2009)
- Michael Collins' NLP course notes (Columbia)

Implementation Resources:

- NLTK library: `https://www.nltk.org/`
- KenLM (fast n-gram toolkit): `https://kheafield.com/code/kenlm/`