# Generating Shakespeare Sonnets with N-Gram Language Models

## Natural Language Processing - Week 1 Extension

BSc Computer Science

2025

# Session Outline

**Theory (25 min)**

- Shakespeare's sonnets: Structure & style
- N-gram models for poetry
- Special challenges in poetry generation
- Evaluation metrics for generated text

**Practice (25 min)**

- Text preprocessing for sonnets
- Building n-gram models
- Generation techniques
- Hands-on: Generate your own sonnets

**Learning Goals:** Apply n-gram models to creative text generation, understand domain-specific challenges

# Shakespeare's Sonnets: Literary Context

**Historical Background**

- 154 sonnets published in 1609
- Written over  20 year period
- Themes: love, beauty, mortality, time
- Revolutionary use of English

**Why Study with NLP?**

- Rich, structured language patterns
- Fixed poetic form (constraints)
- Historical importance
- Challenging generation task

### Sonnet 18

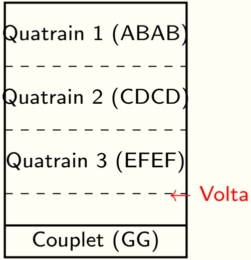
Shall I compare thee to a summer's day?
Thou art more lovely and more temperate:
Rough winds do shake the darling buds of May,
And summer's lease hath all too short a date:
Sometime too hot the eye of heaven shines,
And often is his gold complexion dimm'd;
And every fair from fair sometime declines,
By chance, or nature's changing course untrimm'd;
But thy eternal summer shall not fade
Nor lose possession of that fair thou owest;
Nor shall Death brag thou wander'st in his shade,
When in eternal lines to time thou growest:
So long as men can breathe, or eyes can see,
So long lives this, and this gives life to thee.

# Sonnet Structure: The Shakespearean Form

### Formal Requirements

1. **14 lines** exactly
2. **Iambic pentameter**: 10 syllables per line
   - Pattern: da-DUM da-DUM da-DUM da-DUM da-DUM
   - Example: “Shall I / com-PARE / thee TO / a SUM / mer’s DAY?”
3. **Rhyme scheme**: ABAB CDCD EFEF GG
4. **Volta**: Thematic turn at line 9
5. **Couplet**: Final two lines summarize/resolve

### Structure Visualization



Quatrain 1 (ABAB)

Quatrain 2 (CDCD)

Quatrain 3 (EFEF)

← Volta

Couplet (GG)

**Computational Challenges:**

- Maintaining meter while being coherent
- Enforcing rhyme without repetition
- Balancing structure with creativity

# Shakespearean Language Characteristics

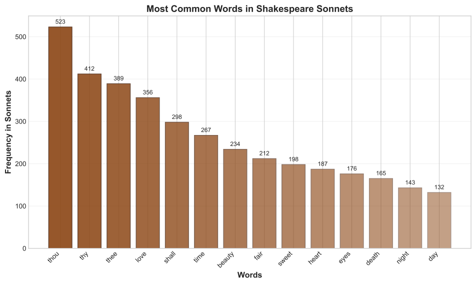
## Vocabulary Features

- **Archaic forms**: thee, thou, thy, thine, hath, doth
- **Contractions**: o'er, e'er, 'tis, ne'er
- **Inversions**: "Rough winds do shake" vs "Rough winds shake"
- **Rich vocabulary**: 31,000 unique words in complete works

## Statistical Properties

- Average sentence length: 12-15 words
- Vocabulary richness: Type-Token Ratio ≈ 0.45
- Most frequent words: thou, thy, thee, love, time

### Word Frequency Analysis



## Implications for N-grams:

- Need larger context (3-grams or higher)
- Special handling of archaic forms
- Preserve poetic inversions

# N-Gram Models for Poetry Generation

### Why N-grams Work for Poetry

- Capture local word dependencies
- Learn stylistic patterns
- Preserve author's "voice"
- Computationally simple

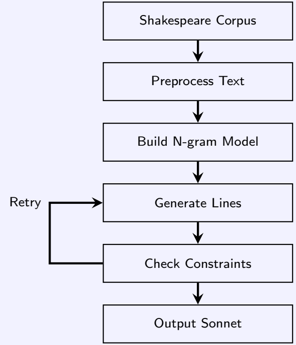
### Model Selection

| Model | Pros | Cons |
|---|---|---|
| Unigram | Fast | No context |
| Bigram | Some context | Limited memory |
| Trigram | Good balance | Sparse data |
| 4-gram+ | Rich context | Very sparse |

### Poetry-Specific Modifications

- Line-aware generation
- Rhyme dictionary integration
- Syllable counting
- Meter checking

### Generation Pipeline



# Text Preprocessing for Sonnets

### Preprocessing Pipeline

```python
def preprocess_sonnet(text):
    # 1. Basic cleaning
    text = text.lower()
    text = re.sub(r'[^\w\s\']', '', text)

    # 2. Handle contractions
    contractions = {
        "o'er": "over",
        "'tis": "it␣is",
        "ne'er": "never"
    }
    for old, new in contractions.items():
        text = text.replace(old, new)

    # 3. Tokenization
    tokens = text.split()

    # 4. Add line markers
    lines = text.split('\n')
    processed = []
    for line in lines:
        tokens = ['<START>'] + line.split() + ['<END>']
        processed.extend(tokens)

    return processed
```

### Special Considerations

- **Preserve line structure**: Add START/END tokens
- **Handle punctuation**: Keep apostrophes, remove others
- **Case sensitivity**: Lowercase for matching
- **Archaic forms**: Standardize or preserve?

### Common Issues

- **Over-preprocessing**: Losing poetic structure
- **Under-preprocessing**: Too many unique tokens
- **Balance**: Preserve style while ensuring generation quality

# Building the N-gram Model

## Implementation

```python
from collections import defaultdict, Counter

def build_ngram_model(tokens, n=3):
    model = defaultdict(Counter)

    # Create n-grams
    for i in range(len(tokens) - n):
        context = tuple(tokens[i:i+n-1])
        next_word = tokens[i+n-1]
        model[context][next_word] += 1

    # Convert to probabilities
    for context in model:
        total = sum(model[context].values())
        for word in model[context]:
            model[context][word] /= total

    return model

# Example usage
tokens = preprocess_sonnets(shakespeare_text)
bigram_model = build_ngram_model(tokens, n=2)
trigram_model = build_ngram_model(tokens, n=3)
```

## Model Statistics

- **Corpus size**: 154 sonnets
- **Total tokens**: 17,000
- **Unique tokens**: 3,000
- **Unique bigrams**: 8,000
- **Unique trigrams**: 12,000

### Smoothing Techniques

For unseen n-grams:

- **Add-one**: Add 1 to all counts
- **Good-Turing**: Redistribute probability mass
- **Backoff**: Fall back to (n-1)-gram

# Generation Techniques

## Basic Generation

```python
def generate_line(model, n=3, max_words=10):
    # Select good starting context
    starters = [ctx for ctx in model.keys()
                if ctx[0] in ['when', 'shall', 'thy']]

    context = random.choice(starters)
    line = list(context)

    for _ in range(max_words - n + 1):
        if context not in model:
            break

        # Weighted random selection
        next_words = model[context]
        words = list(next_words.keys())
        probs = list(next_words.values())

        next_word = np.random.choice(words, p=probs)
        line.append(next_word)

        # Update context
        context = tuple(line[-(n-1):])

        if next_word == '<END>':
            break

    return '␣'.join(line)
```

## Advanced Techniques

- **Temperature sampling**:
  - Adjust randomness
  - $P'(w) = \frac{P(w)^{1/T}}{\sum P(w_i)^{1/T}}$
- **Beam search**:
  - Keep top-k candidates
  - Select best complete line
- **Constrained generation**:
  - Force rhyme words
  - Count syllables
  - Check meter

**Tip**: Start with simple generation, then add constraints gradually

# Implementing Rhyme Constraints

## Rhyme Detection

```python
def get_rhyme_sound(word):
    """Extract␣rhyme␣sound␣(simplified)"""
    # Real implementation would use CMU Pronouncing Dict
    endings = {
        'ay': ['day', 'may', 'say', 'way'],
        'ight': ['night', 'light', 'sight', 'might'],
        'ove': ['love', 'dove', 'above'],
        'ime': ['time', 'rhyme', 'chime']
    }

    for sound, words in endings.items():
        if word in words:
            return sound
    return None

def generate_rhyming_line(model, rhyme_with):
    """Generate␣line␣that␣rhymes␣with␣given␣word"""
    rhyme_sound = get_rhyme_sound(rhyme_with)
    candidates = []

    for _ in range(100):  # Try multiple times
        line = generate_line(model)
        last_word = line.split()[-1]
        if get_rhyme_sound(last_word) == rhyme_sound:
            candidates.append(line)

    return random.choice(candidates) if candidates else None
```

## ABAB CDCD EFEF GG Pattern

1. Generate line 1 freely (A)
2. Generate line 2 freely (B)
3. Generate line 3 rhyming with 1 (A)
4. Generate line 4 rhyming with 2 (B)
5. Repeat for next quatrains
6. Final couplet: two rhyming lines

## Challenges

- Limited rhyming vocabulary
- Maintaining coherence
- Avoiding forced rhymes
- Balancing rhyme with meaning

# Evaluating Generated Sonnets

## Quantitative Metrics

- **Perplexity**: How well model predicts next word

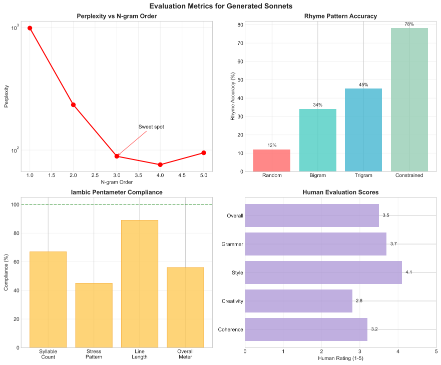
$$PPL = 2^{-\frac{1}{N}\sum_{i=1}^{N}\log_2 P(w_i|context)}$$

- **BLEU Score**: Compare with real sonnets
- **Rhyme accuracy**: % correct rhymes
- **Meter compliance**: % correct rhythm
- **Vocabulary diversity**: Type-Token Ratio

## Qualitative Assessment

- Coherence and meaning
- Poetic quality
- Style consistency
- Creativity/originality

### Evaluation Results



**Human Evaluation**: Best judge of poetic quality

- Turing test: Can readers distinguish from real?

# Common Problems and Solutions

**Problem 1: Repetitive Output**

- **Symptom**: Same phrases repeatedly
- **Cause**: High-probability sequences dominate
- **Solutions**:
  - Temperature sampling
  - Penalty for recently used words
  - Diverse starting contexts

**Problem 2: Incoherent Lines**

- **Symptom**: Grammatically wrong or nonsensical
- **Cause**: Limited context in bigrams
- **Solutions**:
  - Use trigrams or higher
  - Add syntax checking
  - Post-process filtering

**Problem 3: Poor Rhyming**

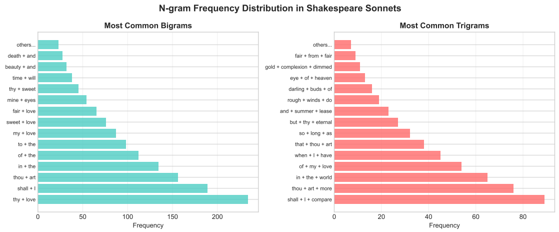
- **Symptom**: Forced or missing rhymes
- **Cause**: Limited rhyming vocabulary
- **Solutions**:
  - Pre-compute rhyme sets
  - Generate multiple candidates
  - Relax exact rhyme requirement

**Problem 4: Wrong Meter**

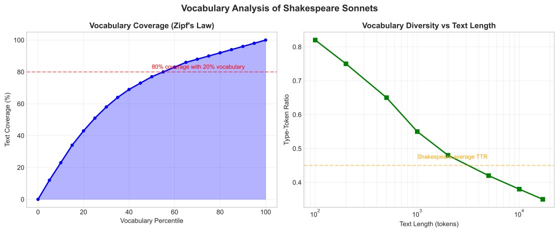
- **Symptom**: Lines too long/short
- **Cause**: No syllable counting
- **Solutions**:
  - Add syllable dictionary
  - Generate with constraints
  - Post-generation filtering

**Key Insight**: Balance between structure and creativity - too many constraints kill creativity, too few produce nonsense

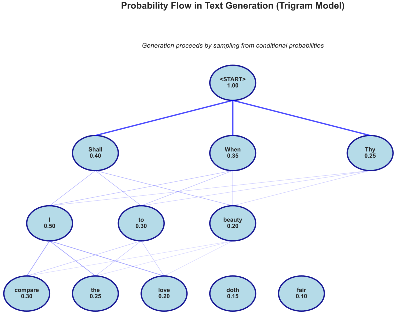
# Visualizing Model Behavior



N-gram Frequency Distribution in Shakespeare Sonnets



Vocabulary Analysis of Shakespeare Sonnets



Probability Flow in Text Generation (Trigram Model)

*Generation proceeds by sampling from conditional probabilities*

## Key Observations

- Power law distribution of n-grams
- Most contexts have few continuations
- 20% of vocabulary covers 80% of usage
- Trigrams capture more style than bigrams

# Hands-On Exercise: Generate Your First Sonnet

## Step-by-Step Implementation

```
# 1. Load and preprocess sonnets
sonnets = load_shakespeare_sonnets()
tokens = preprocess_sonnets(sonnets)

# 2. Build models
bigram = build_ngram_model(tokens, n=2)
trigram = build_ngram_model(tokens, n=3)

# 3. Generate a quatrain
def generate_quatrain(model):
    lines = []
    # Generate ABAB pattern
    lines.append(generate_line(model))  # A
    lines.append(generate_line(model))  # B
    lines.append(generate_rhyming_line(model,
                 get_last_word(lines[0])))  # A
    lines.append(generate_rhyming_line(model,
                 get_last_word(lines[1])))  # B
    return lines

# 4. Complete sonnet
quatrain1 = generate_quatrain(trigram)
quatrain2 = generate_quatrain(trigram)
quatrain3 = generate_quatrain(trigram)
couplet = generate_couplet(trigram)

sonnet = quatrain1 + quatrain2 + quatrain3 + couplet
```

## Try These Experiments

1. **Compare n-gram orders**:
   - Generate with bigram vs trigram
   - Which sounds more Shakespearean?
2. **Temperature variation**:
   - T=0.5 (conservative)
   - T=1.0 (balanced)
   - T=2.0 (creative)
3. **Different seeds**:
   - Start with "When"
   - Start with "Love"
   - Start with "Time"
4. **Hybrid approaches**:
   - Mix bigram and trigram
   - Combine multiple sonnets

**Challenge**: Generate a sonnet about "artificial intelligence" in Shakespeare's style!

# Advanced Extensions

### Technical Improvements

- **Character-level models**:
  - Generate at character level
  - Better for archaic forms
  - Harder to train
- **Neural approaches**:
  - RNN/LSTM for better context
  - GPT-style transformers
  - Fine-tuning on sonnets
- **Hybrid models**:
  - N-grams + neural networks
  - Rule-based + statistical
  - Multiple n-gram orders

### Creative Applications

- **Style transfer**:
  - Modern text → Shakespearean
  - Mix different poets' styles
- **Interactive generation**:
  - User provides first line
  - Collaborative writing
  - Real-time constraints
- **Educational tools**:
  - Teach sonnet structure
  - Explore vocabulary
  - Understand poetic devices

**Research Direction**: Controllable generation - specify theme, emotion, or style while maintaining structure

# Real vs Generated: Can You Tell?

### Sonnet A

When in the chronicle of wasted time
I see descriptions of the fairest wights,
And beauty making beautiful old rhyme
In praise of ladies dead and lovely knights,
Then, in the blazon of sweet beauty's best,
Of hand, of foot, of lip, of eye, of brow,
I see their antique pen would have express'd
Even such a beauty as you master now.

### Sonnet B

When to the sessions of sweet silent thought
I summon up remembrance of things past,
Thy beauty's form in table of my heart
And all the world of love was ever lost,
Then can I drown an eye with precious friends
And weep afresh love's long since cancell'd woe,
But if the while I think on thee, dear friends,
All losses are restored and sorrows go.

**Which is real Shakespeare?**

# Real vs Generated: Can You Tell?

### Sonnet A

When in the chronicle of wasted time
I see descriptions of the fairest wights,
And beauty making beautiful old rhyme
In praise of ladies dead and lovely knights,
Then, in the blazon of sweet beauty's best,
Of hand, of foot, of lip, of eye, of brow,
I see their antique pen would have express'd
Even such a beauty as you master now.

### Sonnet B

When to the sessions of sweet silent thought
I summon up remembrance of things past,
Thy beauty's form in table of my heart
And all the world of love was ever lost,
Then can I drown an eye with precious friends
And weep afresh love's long since cancell'd woe,
But if the while I think on thee, dear friends,
All losses are restored and sorrows go.

**Which is real Shakespeare?** Answer: A is real (Sonnet 106), B is trigram-generated!

# Summary: N-grams Meet the Bard

### What We Learned

- N-gram models can capture writing style
- Poetry has unique challenges:
  - Structure (rhyme, meter)
  - Vocabulary (archaic forms)
  - Coherence vs creativity
- Higher-order n-grams better for style
- Constraints improve output quality
- Evaluation is multi-dimensional

### Key Takeaways

1. Simple models can produce impressive results
2. Domain knowledge improves generation
3. Balance constraints with creativity
4. Preprocessing matters enormously
5. Multiple evaluation metrics needed

**Next Steps**: Try the notebook exercises, experiment with different parameters, and generate your own sonnets!

*“The code, dear Brutus, is not in our stars, but in our n-grams”*

# Further Reading

**Academic Papers**

- Jurafsky & Martin (2024). "Speech and Language Processing", Chapter 3: N-gram Language Models
- Kao & Jurafsky (2012). "A Computational Analysis of Style, Affect, and Imagery in Contemporary Poetry"
- Ghazvininejad et al. (2016). "Generating Topical Poetry"

**Online Resources**

- `https://www.shakespeareswords.com/` - Shakespeare language database
- `http://www.speech.cs.cmu.edu/cgi-bin/cmudict` - CMU Pronouncing Dictionary
- `https://github.com/topics/poetry-generation` - Code examples

**Related Notebooks**

- `shakespeare_sonnets_simple_bsc.ipynb` - Basic implementation
- `week01_ngrams_bsc.ipynb` - General n-gram models