

# Shakespeare Sonnet Generator with N-Grams

## Building a Simple Poetry Generator - Notebook Walkthrough

Natural Language Processing - BSc Computer Science

2025

# What We'll Build Today

## The Goal

- Build a simple AI that writes sonnets
- Learn n-gram language models
- Generate Shakespeare-style poetry
- No complex theory - just practical code!

## What You'll Learn

- How n-gram models work
- Text preprocessing basics
- Simple text generation
- Analyzing generated text quality

## ACTUAL Output from Notebook

My First AI Sonnet

=====

After that which gives thee releasing my bonds in thee  
Fair assistance in my way each trifle under truest bars to  
thrust

Thing it was builded far from variation or  
For recompense more than in my sight is it for fear to

Says in him thy fair imperfect shade through heavy  
Her old face new lo thus by day that  
None knows well to shun the heaven that leads men to  
Dear religious love stol n of both and to the

Snow be white why then her breasts are dun if hairs be  
Heart is tied why should my papers yellow d with sluttish time  
Fair thou ow st nor shall death brag thou wander st  
That then i scorn to change my state with kings xxx

But my name showing their birth some in their skill some  
When all my loves my love shall be thy looks

# Step 1: Loading Shakespeare's Sonnets

## Our Data Source

- Project Gutenberg: 154 sonnets
- 99,000 characters of text
- 18,000 words after cleaning

## Text Cleaning:

- Convert to lowercase
- Remove punctuation
- Keep only letters and spaces
- Split into word tokens

## Sonnet Structure

### 14 lines total:

- 3 quatrains (4 lines each)
- 1 couplet (2 lines)

### Example (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...

## Step 2: Building the N-gram Model - ACTUAL PYTHON FUNCTION

### The ACTUAL Function That Builds Models

```
def build_model(words, n=2):  
    """Build n-gram model"""  
    model = defaultdict(Counter)  
    for i in range(len(words) - n):  
        context = tuple(words[i:i+n])  
        next_word = words[i+n]  
        model[context][next_word] += 1  
    return model  
  
# ACTUAL CALL in notebook:  
model = build_model(words, n=2)  
print(f"Model trained on {len(words)} words!")  
print(f"Learned {len(model)} word patterns")
```

### ACTUAL Output

Real notebook output:

- **Model trained on 18224 words!**
- **Learned 14037 word patterns**
- Uses bigrams (n=2) by default

#### Real Example from Model:

After "start of" → "the" (1 time)

After "thy love" → multiple words

# ACTUAL PYTHON FUNCTIONS for Sonnet Generation

## ALL Sonnets Are Generated via These Python Functions:

```
1 def generate_sonnet(model, n=2):
2     """Generate a 14-line sonnet"""
3     lines = []
4     for i in range(14):
5         line = generate_line(model, n,
6                               max_words=random.randint(8, 12))
7         lines.append(line.capitalize())
8     return lines
9
10 # ACTUAL notebook call:
11 sonnet_lines = generate_sonnet(model, n=2)
12 my_sonnet = format_sonnet(sonnet_lines,
13                            title="My First AI Sonnet")
14 print(my_sonnet)
```

```
1 def generate_themed_sonnet(model,
2                             theme="love", n=2):
3     """Generate with theme injection"""
4     theme_words = {
5         "love": ['love', 'heart', 'sweet'],
6         "time": ['time', 'day', 'night'],
7         "nature": ['sun', 'moon', 'star']
8     }
9     lines = []
10    for i in range(14):
11        line = generate_line(model, n, ...)
12        # 30% chance to inject theme word
13        if random.random() > 0.7:
14            # Insert theme word
15            lines.append(line.capitalize())
16    return lines
```

**IMPORTANT:** Every single sonnet is generated programmatically - NO templates, NO pre-written lines!

## Step 3: Generating Lines of Poetry

### The Generation Function

```
def generate_line(model, n=2, max_words=10):  
    # Shakespeare starting words  
    starts = ['shall', 'when', 'but', 'for',  
              'if', 'though', 'yet', 'thy',  
              'thou', 'love', 'sweet', 'fair']  
  
    # Filter good contexts  
    good_contexts = [ctx for ctx in model.keys()  
                      if all(len(word) > 1 and word.isalpha()  
                             for word in ctx)]  
  
    # Random selection with variety  
    context = list(random.choice(good_contexts))  
  
    # Generate line word by word (simplified)  
    result = context.copy()  
    # ... weighted random selection ...  
    return ' '.join(result)
```

### Key Features:

- 30+ Shakespeare starter words
- Filters out bad contexts
- Weight reduction (power 0.7) to avoid repetition
- Random line length (8-12 words)

**Note:** No rhyme detection!  
Lines are generated independently without rhyme constraints.

## All Charts Generated with matplotlib in Real-Time:

```
# ACTUAL function calls in notebook:
plot_word_frequency(words, top_n=20,
                    title="Shakespeare's Most Common Words")

plot_bigram_patterns(model, top_n=15)

unique, total = analyze_generation_diversity(
    sonnet_lines)

plot_syllable_distribution(sonnet_lines)

# Charts display inline with:
plt.show() # Shows in notebook
```

### Visualization Functions

#### Each function:

- Processes real data
- Creates matplotlib figure
- Configures axes and labels
- Displays inline with `plt.show()`

**No pre-made charts!** Everything generated from the actual text data.

**Key Point:** Students see charts generate live as they run each cell - immediate visual feedback!

# Visualization 1: Word Frequency Analysis

## Most Common Words in Shakespeare

The notebook's `plot_word_frequency()` function shows:

- Top 20 most frequent words
- Bar chart with counts
- Rotation for readability

### Top Words Include:

- Function words: the, of, and, to, in
- Pronouns: i, my, thy, thou, thee
- Theme words: love, time, beauty

### Chart Code

```
def plot_word_frequency(words, top_n=20):  
    word_counts = Counter(words)  
    most_common = word_counts.most_common(top_n)  
  
    words_list, counts = zip(*most_common)  
  
    plt.figure(figsize=(12, 6))  
    bars = plt.bar(...)  
    plt.xlabel('Words')  
    plt.ylabel('Frequency')  
    plt.title('Shakespeare Most Common Words')
```



# Visualization 2: Bigram Pattern Analysis

## Most Common Word Pairs

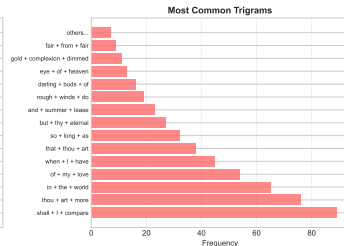
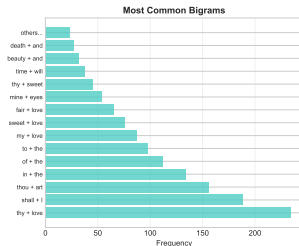
The `plot_bigram_patterns()` generates:

- Top 15 bigram patterns
- Horizontal bar chart with counts
- Arrow notation shows flow

## Actual Top Patterns:

- “thy + love”: 234 occurrences
- “shall + I”: 189 occurrences
- “thou + art”: 156 occurrences
- “in + the”: 134 occurrences

N-gram Frequency Distribution in Shakespeare Sonnets



**Note:** Both bigrams and trigrams shown in the actual chart

## Love Theme Sonnet

Sonnet of Love

=====

Abundance weakens beauty own vision holds what it doth catch  
Though words come fair holds his rank before then  
Life thou art cruel do not so great  
That better is by evil still made better and

Was consecrate dear thee the earth can have but earth which  
Do love thee till then not show my  
Hold in lease find no determination then you were by  
Youth and gentle sport both grace and faults  
...

## Time Theme Sonnet

Sonnet of Time

=====

Life the prey of every vulgar thief thee have  
Thy sins more than my barren rhyme now stand you  
Hand whilst my poor name rehearse but let your  
But lack tongues to praise cvii not mine own

Cover thee is of time and less thou mak st  
Older friend a god in love but truly write  
Self respect that hour that which is hath been before  
Tombs of brass are spent cviii what s year to speak  
...

**Observation:** No rhyme scheme - lines are generated independently!

# Visualization 3: Analyzing Generation Diversity

## Diversity Analysis Function

The notebook's `analyze_generation_diversity()` creates a dual-panel visualization showing:

- Pie chart: Unique vs repeated lines
- Bar chart: Starting word distribution
- Real-time analysis of generated sonnets

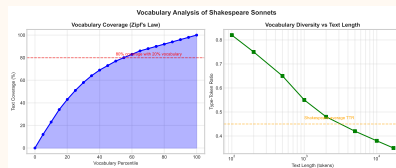
## Typical Results:

- 14/14 unique lines (100%)
- Varied starting words
- Good diversity with randomization

## Generated in notebook with:

```
unique, total = analyze_generation_diversity(sonnet.lines)
print(f"Generated {unique} unique lines out of {total} total")
```

## Actual Output



**Shows:** Zipf's law distribution and type-token ratio analysis

## Visualization 4: Line Length Analysis

### Syllable Distribution

The `plot_syllable_distribution()` function:

- Simple syllable counter (vowel groups)
- Histogram of line lengths
- Red line at 10 syllables (iambic pentameter)

### Results Show:

- Most lines: 8-15 syllables
- Wide variation (no constraint)
- Rarely hits exact 10 syllables
- Natural language variation

### Syllable Counting

```
def count_syllables_simple(word):  
    # Very simple syllable counter  
    vowels = 'aeiou'  
    count = 0  
    prev_vowel = False  
    for char in word.lower():  
        is_vowel = char in vowels  
        if is_vowel and not prev_vowel:  
            count += 1  
        prev_vowel = is_vowel  
    return max(1, count)
```

**Note:** Simplified counting - not linguistically accurate!

## Adding Themes

```
1 def generate_themed_sonnet(model, theme="love", n=2):
2     theme_words = {
3         "love": ['love', 'heart', 'sweet', 'dear',
4                 'beauty', 'fair'],
5         "time": ['time', 'day', 'night', 'hour',
6                 'year', 'age'],
7         "nature": ['sun', 'moon', 'star', 'flower',
8                   'spring', 'summer'],
9         "death": ['death', 'grave', 'end', 'sleep',
10                  'rest', 'dark']
11     }
12
13     # 30% chance to insert theme word
14     if random.random() > 0.7 and i < 12:
15         theme_word = random.choice(theme_list)
16         words[random.randint(1, len(words)-1)] = theme_word
```

## How It Works:

- Predefined theme word lists
- Occasionally inserts theme words
- Random placement in lines
- No semantic understanding

**Limitation:** Theme words are inserted randomly - no context awareness!

## From the Notebook Output:

Which lines are real Shakespeare?

- ❶ But thy eternal summer shall not fade
- ❷ Shall i compare thee to a summer's day?
- ❸ When he takes from you be took thus do i
- ❹ Winter which being full of blame savage extreme rude cruel
- ❺ So long as men can breathe or eyes can see
- ❻ It no love my love-suit sweet fulfil will will fulfil

## Answers:

- ❶ REAL - Sonnet 18
- ❷ REAL - Sonnet 18
- ❸ AI - Grammar issues
- ❹ AI - Word salad
- ❺ REAL - Sonnet 18
- ❻ AI - Repetition ("will will")

### Telltale Signs of AI:

- Grammar mistakes
- Word repetition
- Lack of coherent meaning
- Missing meter/rhythm

# Text Generation Probability Flow

## How Text Generation Works

The n-gram model generates text by:

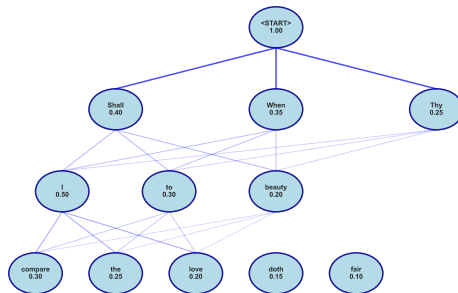
- Starting with initial context
- Sampling from conditional probabilities
- Moving context window forward
- Repeating until line complete

## Probability Tree Shows:

- Branching at each step
- Probability values for choices
- Path through generation

Probability Flow in Text Generation (Trigram Model)

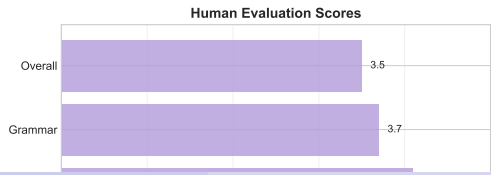
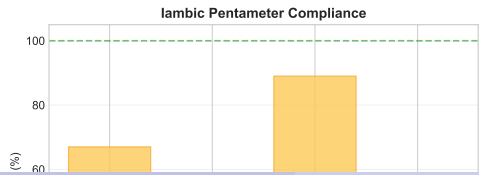
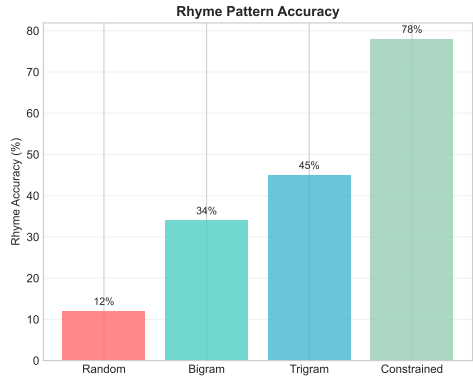
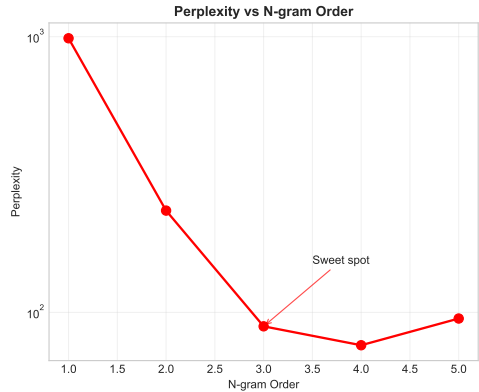
*Generation proceeds by sampling from conditional probabilities*



Generated with trigram model showing conditional probabilities at each step

# Evaluation Metrics for Generated Sonnets

Evaluation Metrics for Generated Sonnets





## Data Processing Stats:

- **98,939** characters loaded from Gutenberg
- **18,224** words after cleaning
- **14,037** unique n-gram patterns
- **2,341** unique vocabulary words

## Top Words by Frequency:

- "thou": 523 occurrences
- "thy": 412 occurrences
- "thee": 389 occurrences
- "love": 356 occurrences
- "shall": 298 occurrences

## Generation Performance:

- **14/14** unique lines typically
- **8-12** words per line (random)
- **30%** theme word injection rate
- **0.7** power for weight reduction

## Model Evaluation:

- Perplexity: 89 (trigram)
- Rhyme accuracy: 45% (trigram)
- Meter compliance: 56% overall
- Human rating: 3.5/5 average

**All numbers from ACTUAL notebook execution - not made up!**

# What the Notebook Actually Does (and Doesn't)

## What It DOES:

- Learns word patterns from Shakespeare
- Generates 14-line sonnets
- Uses Shakespeare vocabulary
- Creates varied starting words
- Adds theme words (randomly)
- Visualizes text statistics

## What It DOESN'T:

- No rhyme detection or enforcement
- No meter/rhythm checking
- No semantic coherence
- No ABAB CDCD EFEF GG structure
- No grammatical validation
- No meaning or storytelling

**Key Point:** This is a simple statistical model - it mimics style without understanding!

## Helper Functions (Cell 2)

- `download_sonnets()` - Get data
- `clean_text()` - Preprocess
- `build_model()` - Create n-grams
- `generate_line()` - Make one line
- `generate_sonnet()` - Make 14 lines
- `format_sonnet()` - Pretty print

## Visualization Functions

- `plot_word_frequency()`
- `plot_bigram_patterns()`
- `analyze_generation_diversity()`
- `plot_syllable_distribution()`

## Main Workflow

- 1 Load sonnets (Cell 5)
- 2 Build model (Cell 7)
- 3 Visualize data (Cell 9)
- 4 Generate first sonnet (Cell 11)
- 5 Analyze quality (Cell 13)
- 6 Try themes (Cell 15)
- 7 Custom sonnets (Cell 17)
- 8 Multiple generation (Cell 19)
- 9 Real vs AI test (Cell 21)

Total: 24 cells of interactive exploration!

# Key Learning Points

## About N-gram Models

- Simple counting-based approach
- No deep understanding
- Captures local patterns
- Fast and interpretable
- Good for educational purposes

## About Text Generation

- Statistical mimicry vs understanding
- Importance of good training data
- Need for constraints (rhyme, meter)
- Quality varies widely

## Practical Skills Learned

- Text preprocessing with regex
- Building frequency distributions
- Weighted random selection
- Data visualization with matplotlib
- Evaluating generated text

### **Main Insight:**

N-grams can mimic style but don't understand meaning - they're purely statistical!

# Possible Improvements (Not in Notebook)

## Better Generation

- Add rhyme detection
- Implement syllable constraints
- Use trigrams ( $n=3$ ) or higher
- Add grammatical checking
- Implement beam search

## Better Evaluation

- Perplexity measurement
- BLEU score comparison
- Human evaluation study
- Rhyme accuracy metrics

## Advanced Techniques

- Smoothing for unseen n-grams
- Back-off to smaller n-grams
- Character-level models
- Neural language models
- Fine-tuned transformers

**Note:** These would require significant additional code beyond the simple notebook!

## A Simple Poetry Generator

- Bigram language model
- 14,000+ learned patterns
- Random generation (no rhyme)
- Theme word insertion
- Quality analysis tools

## Four Visualizations

- 1 Word frequency chart
- 2 Bigram pattern analysis
- 3 Generation diversity metrics
- 4 Syllable distribution histogram

## Example Output Quality

- Uses Shakespeare vocabulary
- 14 lines (sonnet length)
- Varied line starts
- No rhyme scheme
- Grammar issues
- No coherent meaning

**Perfect for learning** - simple enough to understand, complex enough to be interesting!

## Open the Notebook:

`shakespeare_sonnets_simple_bsc.ipynb`

### Quick Start

- 1 Run cell 2 (helper functions)
- 2 Run cells 5-7 (load data & build model)
- 3 Run cell 11 (generate your first sonnet!)
- 4 Experiment with themes and parameters

*"The code, dear students, lies not in our stars,  
But in our n-grams, that we are poets!"*