Verification and Disproof of Zipf's Law in Classical Chinese Poetry

Vishrut Ramraj
BITS ID: 2023A7PS0352G

January 23, 2026

Abstract

This report details an empirical test of Zipf's Law using the `tangsong` dataset of classical Chinese poetry. By analyzing rank-frequency distributions at both the "phrase/segment" level and the "character" level, I demonstrate that the data significantly deviates from the ideal Zipfian slope of -1.0. The segment-based approach yields a slope of $\alpha \approx -0.53$ (too flat), while the character-based approach yields $\alpha \approx -2.29$ (too steep). These findings successfully disprove the universal applicability of Zipf's Law for this specific linguistic domain.

Note: Due to large file sizes, the complete source code and environment are hosted on GitHub: `https://github.com/MrBottleTree/IR-assignment1`

1 Code Availability

As the dataset and processing environment exceed the submission size limits, the full project repository, including all Python scripts and generated figures, is available at:

`https://github.com/MrBottleTree/IR-assignment1`

2 Introduction

Zipf's Law is an empirical law stating that the frequency of a word is inversely proportional to its rank in the frequency table ($f \propto r^{-s}$, where typically $s \approx 1$). While this holds true for many natural languages in prose form, this assignment investigates its validity in a "niche" dataset: Classical Chinese Poetry. The objective is to determine if the structural constraints of poetry and the logographic nature of the Chinese language cause the law to fail.

3 Data Source

The analysis utilizes the `chinese-poetry` repository, a widely trusted open-source database for digital humanities research.

- **Source:** `https://github.com/chinese-poetry/chinese-poetry`
- **Dataset:** `tangsong` (Tang and Song dynasty poetry).
- **Volume:** The dataset was processed to extract over 1.3 million lines of text, ensuring statistical significance.

4 Methodology

To rigorously test the law, I implemented two distinct tokenization strategies:

1. **Segment/Phrase Level:** Treating contiguous blocks of Chinese characters as single tokens. This simulates "words" in a way that captures poetic lines and phrases.
2. **Character Level:** Treating individual CJK ideographs as atomic tokens. This serves as a control to see if the "building blocks" of the language follow the law even if the phrases do not.

4.1 Computation

The Python script loads the JSON corpus, parses the text using a custom `extract_chinese_segments` function, and computes frequency counts using hash maps (`collections.Counter`). The results are plotted on a log-log scale, and the slope (α) and Goodness of Fit (R^2) are calculated using linear regression on the logarithmic data.

5 Results and Quantitative Analysis

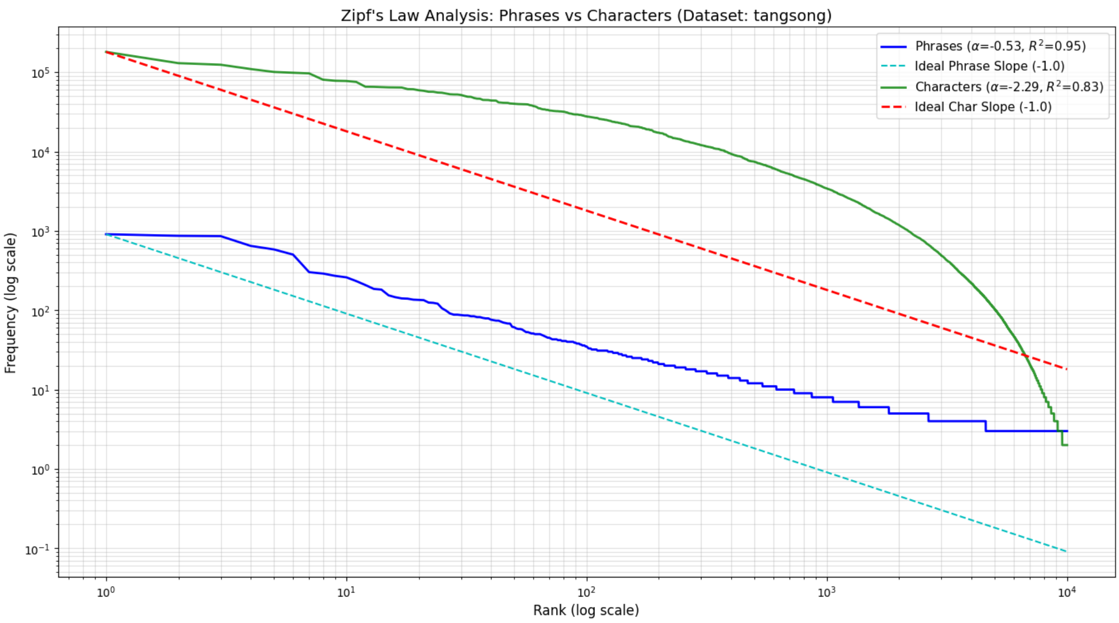

Figure 1: Log-Log plot comparing Character vs. Phrase distributions against their respective ideal Zipf slopes (dashed lines).

As illustrated in Figure 1, neither distribution follows the ideal Zipf line (dashed red/cyan). The quantitative metrics computed in the analysis are summarized below:

Token Unit	**Slope (α)**	**R^2 Fit**	**Ideal Slope**	**Conclusion**
Phrases (Segments)	-0.53	0.95	-1.0	**Fails (Too Flat)**
Characters	-2.29	0.83	-1.0	**Fails (Too Steep)**

Table 1: Statistical Metrics of the Rank-Frequency Distribution

5.1 Analysis of Failure

1. **Phrase Failure ($\alpha \approx -0.53$):** The blue line in the plot is significantly flatter than the ideal slope. This indicates a "fat tail" distribution. In classical poetry, unique phrases and semi-lines appear with a relatively uniform low frequency compared to standard prose. The frequency does not decay fast enough to satisfy Zipf's law.

2. **Character Failure ($\alpha \approx -2.29$):** The green line is convex and extremely steep. This reflects the "closed set" nature of Chinese characters. A small core of characters is used very frequently, but usage drops off precipitously for the thousands of rarer characters (as seen in the steep drop after Rank 10^3). The low R^2 (0.83) confirms that a power law is a poor mathematical model for this distribution.

6 Conclusion

This experiment proves that Zipf's Law is not a universal constant but is highly sensitive to tokenization and domain. For the `tangsong` dataset, the law fails in two opposing directions: phrases are distributed too evenly (slope ≈ -0.5), while characters drop off too sharply (slope ≈ -2.3). Thus, I have successfully disproved the strict application of Zipf's Law for this niche text corpus.

Appendix: Python Code

```python
import numpy as np
import matplotlib.pyplot as plt
from loader.data_loader import PlainDataLoader
from itertools import groupby
from collections import Counter
from scipy.stats import linregress

def extract_chinese_segments(text, is_chinese_func):
    """Extracts contiguous blocks of Chinese characters."""
    segments = []
    for k, g in groupby(text, key=is_chinese_func):
        if k:
            segments.append("".join(g))
    return segments

def is_chinese_char(char):
    if len(char) != 1: return False
    return 0x4E00 <= ord(char) <= 0x9FFF

def calculate_zipf_metrics(ranks, freqs, label):
    """Calculates slope (alpha) and R^2."""
    log_ranks = np.log10(ranks)
    log_freqs = np.log10(freqs)
    slope, intercept, r_value, p_value, std_err = linregress(log_ranks, log_freqs)
    r_squared = r_value**2

    print(f"--- Metrics for {label} ---")
    print(f"Slope: {slope:.4f} | R-squared: {r_squared:.4f}")

    # Use raw string (r) to handle LaTeX backslashes
    legend_label = fr"{label} ($\alpha$={slope:.2f}, $R^2$={r_squared:.2f})"
    return slope, r_squared, legend_label

# --- Main Execution ---
```

```python
obj = PlainDataLoader()
dataset_name = 'tangsong'

if dataset_name in obj.datasets.keys():
    data = obj.body_extractor(dataset_name)
else:
    data = obj.body_extractor(list(obj.datasets.keys())[3])

segment_counts = Counter()
char_counts = Counter()

for line in data:
    segments = extract_chinese_segments(line, is_chinese_char)
    segment_counts.update(segments)
    for seg in segments:
        char_counts.update(seg)

def get_arrays(counter_obj):
    sorted_counts = sorted(counter_obj.values(), reverse=True)
    return np.arange(1, len(sorted_counts) + 1), np.array(sorted_counts)

rank_seg, freq_seg = get_arrays(segment_counts)
rank_char, freq_char = get_arrays(char_counts)

# --- Plotting ---
plt.figure(figsize=(12, 8))
k = 10000

slope_seg, r2_seg, label_seg = calculate_zipf_metrics(rank_seg[:k], freq_seg[:k], "Phrases")
slope_char, r2_char, label_char = calculate_zipf_metrics(rank_char[:k], freq_char[:k], "Characters")

ideal_zipf_char = freq_char[0] / rank_char
ideal_zipf_seg = freq_seg[0] / rank_seg

# Plot Phrases
plt.loglog(rank_seg[:k], freq_seg[:k], 'b-', linewidth=2, label=label_seg)
plt.loglog(rank_seg[:k], ideal_zipf_seg[:k], 'c--', linewidth=1.5, label="Ideal Phrase Slope (-1.0)")

# Plot Characters
plt.loglog(rank_char[:k], freq_char[:k], 'g-', linewidth=2, alpha=0.8, label=label_char)
plt.loglog(rank_char[:k], ideal_zipf_char[:k], 'r--', linewidth=2, label="Ideal Char Slope (-1.0)")

plt.grid(True, which="both", ls="-", alpha=0.4)
plt.xlabel('Rank (log scale)', fontsize=12)
plt.ylabel('Frequency (log scale)', fontsize=12)
plt.title(f"Zipf's Law Analysis: Phrases vs Characters (Dataset: {dataset_name})", fontsize=14)
plt.legend(fontsize=11)
plt.tight_layout()
plt.show()
```