Statistical Hangman Solver

Ashutosh Rabia, Indian Institute of Technology Kanpur

10th May 2025

Introduction

Hangman is a word-guessing game that challenges players to deduce a hidden word by guessing individual letters. The game typically involves two participants: one player who thinks of a word and another player who tries to guess it. The word to be guessed is represented by a series of dashes, each dash representing a letter in the word. The guessing player's task is to sequentially suggest letters they believe are in the word. The game continues until the guessing player either successfully guesses the entire word or the guessing player has made 6 incorrect guesses, indicating a loss. The guessing player can use strategies based on the frequency of letters in the English language, common letter combinations, and other linguistic patterns to narrow down potential letters and maximize the chances of guessing the word correctly within a limited number of attempts.
This report documents a high performance solver that:

- Preprocesses a 250 000 word dictionary with boundary markers.
- Builds frequency tables for n-grams of order 1 through 5.
- Applies a convolutional-style scoring over masked patterns to select the next letter.

Intuition

The solver leverages linguistic patterns and conditional probabilities:

- Common substrings (e.g., "QU", "ING", "TION") appear frequently in English.
- Word boundaries (start: '{ ', end: '|') help identify suffixes and prefixes.
- Longer n-grams provide stronger signals when more context is revealed.
- We combine n-gram orders with weights $\{1, 1, 4, 10, 20\}$ for unigrams through 5-grams.

Methodology

Preprocessing

Dictionary preprocessing involves:

1. **Load words:** Read and lowercase all entries from the 250 000 word list.
2. **Frame words:** Surround each word `w` with "{w|" to mark boundaries in n-gram extraction.
3. **Build n-gram tables:** For each $n = 1 \dots 5$, allocate a NumPy array of shape $(28)^n$ (26 letters plus two boundary symbols) and increment the count for every contiguous n-gram in each framed word.

Inference

At each guess step given a masked pattern (e.g., "$_p p_e$") :

Clean and frame: Strip spaces and frame as "{pattern|".

One-hot encode: Construct a $26 \times L$ binary matrix X marking known letters at their positions.

Compute scores:

- *Unigrams:* Multiply each letter's total frequency by the number of known occurrences in X.
- *Higher-order grams:* For each $n = 2 \ldots 5$, treat the n-gram table as a 1D convolution filter. Slide a window of size n over the framed pattern; when exactly one blank appears, replace it with each candidate letter, look up the n-gram count, normalize by excluding guessed letters, weight by β_n, and accumulate.

Exclude guesses: Zero out scores for letters in the exclusion mask.

Select guess: Choose the letter with the highest aggregated score.

Data Structures

Frequency tables use the following shapes:

- **Unigram:** (28) vector (a–z, "{", "|"),
- **Bigram:** (28×28) matrix,
- **Trigram:** $(28 \times 28 \times 28)$ tensor,
- **4-gram:** 4D tensor, 5-gram: 5D tensor of shape $(28)^4$ and $(28)^5$ respectively.

Code Snippet

Below is a simplified excerpt of the `guess()` method illustrating the core scoring loop:

```
# scores: accumulate weighted n-gram responses
scores = np.zeros(28)
weights = [1,1,4,10,20]
for order in range(1,6):
    table = self.freq_tables[order-1]
    w = weights[order-1]
    for i in range(len(framed)-order+1):
        segment = framed[i:i+order]
        if segment.count('_')==1:
            cnt = np.zeros(28)
            for c in 'abcdefghijklmnopqrstuvwxyz':
                cand = segment.replace('_', c)
                idxs = tuple(self._char_to_idx(ch) for ch in cand)
                cnt[ord(c)-97] = table[idxs]
            scores += self._normalize(cnt) * w
# select highest-scoring letter
letter = chr(np.argmax(scores) + ord('a'))
```

Function Descriptions

Function	Description
_select_url	Returns the Hangman API endpoint URL.
_load_dict	Reads and lowercases the dictionary file into a list of words.
_init_freq_tables	Initializes empty NumPy arrays for n-gram orders 1–5.
_char_to_idx	Maps a character (a–z, '{', '|') to its array index (0–27).
_compute_freq_tables	Iterates over every framed word to populate n-gram frequency counts.
_normalize	Applies the exclusion mask and normalizes an n-gram count vector.
guess	Executes convolutional-style scoring of masked patterns to choose the next best letter.

Table 1: Summary of key HangmanAPI methods

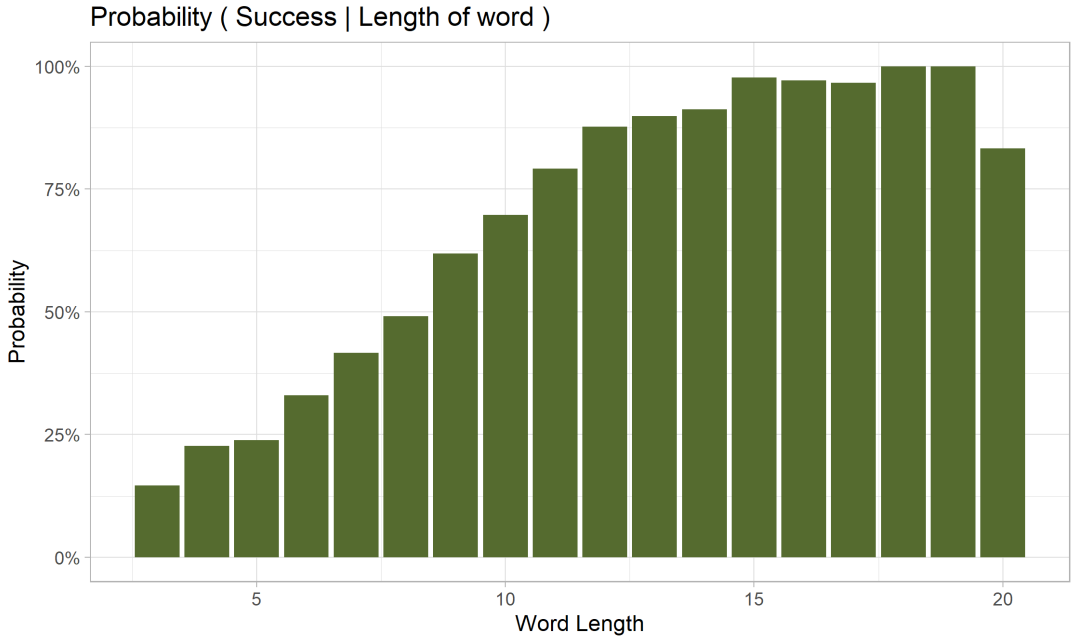

Figure 1: Probabilities of successfully guessing the the word of a given length

Conclusion

I presented a robust Hangman solver combining precomputed n-gram tables up to order 5 with a convolutional inference step, achieving high accuracy 61% . Future work may explore adaptive weighting and neural-statistical hybrids.