Hangman Solver: Intelligent Guessing with Multi-Strategy Decision Making

Sumit Prabhakar

April 2025

Abstract

This report presents the core logic behind an intelligent Hangman solver. The solver employs a multi-strategy algorithm that integrates information theory, statistical pattern recognition, and heuristic-based guessing. The core decision-making function, `guess()`, adapts dynamically to game progress and leverages structural and frequency-based features to efficiently identify the target word.

1 Function Overview: `guess()`

The `guess()` method serves as the core decision engine of the solver. It takes two inputs:

- **word**: A string with known and unknown letters (e.g., `_e__o`).
- **guessed_characters**: A set containing previously guessed letters.

Its objective is to return the most probable next character to guess.

2 Dictionary Filtering

The function first filters the dictionary to retain only those words that match the structure of the input. It converts the word into a regular expression (e.g., `_e__o` $\rightarrow$ `.e..o`) and matches it against the internal word list.

```
pattern = re.compile("^" + clean_word + "$")
filtered_dictionary = [w for w in self.dictionary if pattern.match(w)]
```

3 Scoring Strategies

To evaluate each unguessed character, the function uses four distinct scoring strategies:

3.1 Entropy-Based Scoring

Inspired by information theory, this strategy simulates the effect of guessing each letter and partitions the remaining words accordingly. It calculates the expected information gain (entropy) based on the size of each resulting group.

3.2 Digram Frequency Scoring

Analyzes adjacent letter frequencies relative to known characters. For instance, if "e" is known at position 2, the score of letters adjacent to "e" is increased if they occur frequently in that context.

3.3 Position-Specific Frequency

For each unknown position, the function calculates how frequently each candidate letter appears across all remaining possible words. Letters that appear often in specific positions are prioritized.

3.4 Overall Frequency Score

A general-purpose fallback that scores letters based on how often they occur in the remaining word list.

4 Dynamic Weighting

The final score for each letter is computed as a weighted sum of the four components:

$$\text{Final Score} = 0.3 \cdot \text{Entropy} + 0.1 \cdot \text{Digram} + 0.3 \cdot \text{Position} + 0.3 \cdot \text{Frequency} \tag{1}$$

Weights are chosen to balance early-game exploration with late-game precision.

5 Fallback Mechanisms

If no dictionary matches remain, the solver resorts to:

- Word frequency priors (e.g., suffix patterns like "ing", "ed").
- Substring-based corpus matching.
- Global letter frequency distributions.

These ensure graceful degradation in sparse-data scenarios.

6 Helper Modules

The `guess()` function leverages several utility functions:

- `load_word_frequencies()`: Loads word frequency scores from a corpus.
- `get_position_specific_frequencies()`: Builds positional frequency tables.
- `func()`, `func2()`: Heuristic substring matchers.

7 Conclusion

The Hangman solver demonstrates a hybrid decision-making framework that integrates:

- Data filtering using regex.
- Entropy maximization for informed guessing.
- Contextual linguistic features like digrams and position-wise analysis.
- Adaptive scoring and fallback heuristics.

Such an approach mirrors real-world applications like predictive text input, OCR correction, and NLP-based decoding tasks.

Appendix

Sample Output

```
> Word: _ e _ _ o
> Guessed: {'e', 'a', 's', 't'}
> Guess(): 'r'
```