Hangman N-gram Strategy — Technical Documentation

Sudhanshu Kumar Singh — IIT Guwahati

Contents

Executive Summary (TL;DR)

- Build character n-gram frequency tables (1–5 letters) from a large training dictionary.
- For each turn, parse the masked word (e.g., `_ p p _ e `) into a clean form (e.g., `_pp_e`) and extract windows that contain **exactly one blank** (`_`).
- For every candidate letter, aggregate evidence across 5-gram → unigram with fixed blending weights:

$$P(\ell \mid \text{mask}) = 0.40\, P_5(\ell) + 0.25\, P_4(\ell) + 0.20\, P_3(\ell) + 0.10\, P_2(\ell) + 0.05\, P_1(\ell).$$

- Exclude already guessed letters; when running low on tries, **recalibrate** by removing words containing proven wrong letters and rebuilding n-grams.
- Choose the letter with highest blended probability; fallback prefers vowels if no evidence exists.
- Use a resilient HTTP client (latency probing, retries, backoff) to interact with the game server.

1 System Overview

1.1 Architecture

```
+--------------------+     build_n_grams()      +--------------------------+
| Training Dictionary| -----------------------> | N-gram Stores (1..5-gram) |
+--------------------+                           +------------+--------------+
                                                              |
             API (/new_game, /guess_letter)                   | probabilities
+---------------------+  masked word + tries   +-------------v----------------+
| Game Server (HTTP)  | <--------------------> | HangmanAPI.guess(word_mask) |
+----------+----------+                         +-----------------------------+
           ^                                                 |
           |                       choose letter             v
    requests + backoff                               return best next guess
```

1.2 Main Components

- **Dictionary ingestion**: loads `/kaggle/input/dataset002/words_250000_train.txt`.
- **N-gram builder**: constructs nested counters for 1–5 grams.
- **Guess engine**: evaluates candidate letters from 5-gram down to unigram.
- **Adaptive filtering**: rebuilds n-grams excluding letters known to be wrong.
- **HTTP client**: selects fastest base URL, interacts with `/trexsim/hangman`, retries with exponential backoff.

2 Data & Preprocessing

2.1 Dictionary

- **Path**: `/kaggle/input/dataset002/words_250000_train.txt`.
- **Format**: one lowercase word per line (letters a–z assumed).
- **Letter set**: `letter_set = sorted(set("".join(full_dictionary)))`.

2.2 Mask Format (Game State)

- Server mask has spaces between characters, e.g., `_ p p _ e` .
- Clean representation uses every second char: `clean = word[::2]` ⇒ `_pp_e`.
- Underscore `_` denotes unknown letters.

3 N-gram Model Construction

3.1 Data Structures

- **Unigram**: `unigram[len_word][letter] -> count` (counts unique letters per word).
- **Bigram**: `bigram[len_word][c1][c2] -> count` (conditioned on word length).
- **Trigram**: `trigram[c1][c2][c3] -> count`.
- **Four-gram**: `fourgram[c1][c2][c3][c4] -> count`.
- **Five-gram**: `fivegram[c1][c2][c3][c4][c5] -> count`.

3.2 Counting Logic

- Slide a 5-char window across each word to update 5-grams; populate trailing 4/3/2-grams, especially for short words.
- For words of length 2–3, only applicable lower-order grams are updated.
- For words of length ≥ 4, ensure tail bigrams/trigrams/fourgrams are populated.
- Unigrams add each *distinct* letter once per word using `set(word)`.

Rationale. Conditioning bigrams/unigrams on word length captures different transition patterns in short vs. long words. Using `set(word)` for unigrams emphasizes presence rather than repetition.

4 Guessing Strategy (Per Turn)

4.1 High-Level Flow

1. Track wrong letters: `incorrect = guessed_letters - letters_in_mask`.
2. Adaptive prune: if `tries_remaining <= 5`, rebuild n-grams from dictionary words that **exclude** wrong letters.
3. Initialize a zero probability vector over the alphabet.
4. Apply n-gram passes: 5-gram → 4-gram → 3-gram → 2-gram → unigram.
5. For each pass k, compute $P_k(\ell)$ from windows with **exactly one** blank and blend:

$$\text{prob} w_k \cdot P_k, \quad w = (0.40, 0.25, 0.20, 0.10, 0.05).$$

6. Normalize (optional) and choose the highest-probability letter not yet guessed.
7. Fallback: prefer vowels `e,a,i,o,u`, then others in random order.

4.2 What Each Pass Looks For

Only windows with **exactly one `_`** contribute.

5-gram pass (size 5). At index $i \in [0, \text{len} - 5]$:

`LLLL_`, `LLL_L`, `LL_LL`, `L_LLL`, `_LLLL`.

4-gram pass (size 4).

`LLL_`, `LL_L`, `L_LL`, `_LLL`.

3-gram pass (size 3).

`LL_`, `L_L`, `_LL`.

2-gram pass (size 2).

`L_`, `_L`

(bigrams are conditioned on word length).

Unigram pass. For each `_`, add `unigram[len(word)][letter]` for all candidate letters.

4.3 Blending & Decision Rule

Weights prioritize longer-range structure (morphology, prefixes/suffixes) while still leveraging shorter contexts when longer ones are absent. Final decision is $\arg\max_\ell \text{prob}[\ell]$ over unguessed letters.

5 Adaptive Re-calibration (Pruning)

When `tries_remaining <= 5`, rebuild the n-grams from the subset of dictionary words that **exclude** any letter in `incorrect_guesses`.

- **Effect**: narrows the search space and sharpens statistics as evidence accumulates.
- **Trade-off**: rebuilding is roughly $O(|\text{dict}| \cdot \text{avg\_len})$; do sparingly.

6 Alternate Heuristics (Optional)

6.1 Frequency-Only Fallback (`Originalguess`)

Filter the dictionary by the regex form of the current mask (convert `_` to `.`). Pick the most frequent letter in the remaining candidates; fallback to global frequency if no match.

6.2 Relative Position Heuristic (`relative_guess`)

When `tries_remains == 1` and a specific pattern is observed (e.g., first char blank while third is known), estimate the first letter from conditional frequencies.
Scope fix: replace `for i in full_dictionary:` with `for i in self.full_dictionary:`.

7 API Client & Reliability

7.1 Base URL Selection

Probe candidate links (e.g., `https://trexsim.com`) several times; select the lowest observed latency; append `/trexsim/hangman`.

7.2 Endpoints

- `GET /new_game?practice={bool}` → {`game_id`, `word`, `tries_remains`, ...}
- `POST /guess_letter` with {`game_id`, `letter`} → {`ongoing`success|failed|
- `GET /my_status` → account/game stats

7.3 Request Layer

- Adds `access_token` if present.
- Exponential backoff on `ConnectionError`/`Timeout` (up to 10 retries: 1s, 2s, 4s, ...).
- Parses JSON and raises a normalized `HangmanAPIError` on server errors.
- Note: `verify=False` skips TLS verification; enable `verify=True` in production.

8 Worked Example

Mask: `_ p p _ e` ⇒ clean `_pp_e` (length 5).

1. 5-gram: only windows with a single blank (e.g., `_pp_e` matches the case `\_LLLL` at start).
2. 4-gram: only windows with exactly one `_` contribute (skip those with two blanks).
3. 3-gram: evaluate `_pp`, `pp_`, `p_e`.
4. 2-gram: evaluate `_p` and `p_` (skip `pp`).
5. 1-gram: two blanks contribute unigram evidence twice.
6. Blend with weights and choose the maximum.

9 Complexity & Performance

- N-gram build: $\mathcal{O}(N \cdot L)$ where N is #words and L is average length.
- Per-turn scoring: $\mathcal{O}(L \cdot |\Sigma|)$ across passes (alphabet typically $|\Sigma| = 26$).
- Memory: nested sparse maps for 1–5 grams; consider pickling or trie compression if needed.

10 Configuration & Environment

- Python 3.8+; dependencies: `requests`, `collections`, `random`, `time`, `re`, `urllib.parse` (stdlib).
- Ensure the dictionary file path exists; make path configurable outside Kaggle.

Recommended knobs:

- Weights: $(0.40, 0.25, 0.20, 0.10, 0.05)$.
- Recalibration threshold: `tries_remaining <= 5`.
- Minimum-evidence checks to avoid divide-by-zero.

11 Limitations & Failure Modes

- Out-of-vocabulary words (proper nouns, hyphenations) may be missed.
- Early sparse masks yield thin evidence; multiple blanks per window are intentionally ignored.
- Using `set(word)` for unigrams downweights repeated letters; rely on higher n-grams for double-letter patterns.
- Security: avoid `verify=False` outside of controlled environments.

12 Testing & Debugging Tips

- Seed RNG when using vowel-biased fallback to reproduce runs.
- Log per-pass totals (`total_count`) to see which level dominates.
- Sanity checks:
  - `incorrect_guesses` only tracks letters *not* visible in the current mask.
  - `guessed_letters` updates after each server response.
  - Window filters enforce exactly one `_` per counted window.

13 Suggested Improvements (Roadmap)

1. **Regex pre-filter in main pipeline**: intersect dictionary with current mask before counting n-grams.
2. **Smoothing**: Laplace (add-k) or Katz backoff to reduce zeros.
3. **Positional priors**: position-specific letter distributions per word length.
4. **Learnable weights**: optimize blending via cross-validation on historical games.
5. **Caching**: memoize per-mask probability vectors.
6. **Entropy-based control**: use probability entropy to trigger pruning or heuristic switches.

14 Example Pseudocode (Core Loop)

Listing 1: Core scoring and selection loop (pseudocode)

```
mask = "_ p p _ e "
clean = mask[::2]  # "_pp_e"
prob = zeros(|Sigma|)

for k, w in [(5,0.40), (4,0.25), (3,0.20), (2,0.10), (1,0.05)]:
    Pk = zeros(|Sigma|)
    for window in sliding_windows(clean, size=k):
        if count('_' in window) != 1:
            continue
        anchors, blank_idx = extract(window)
        for letter in Sigma - guessed_letters:
            cnt = count_kgram_match(k, anchors, letter, blank_idx)  # from n-grams
            if cnt > 0:
                Pk[letter] += cnt
    if sum(Pk) > 0:
```

```
        Pk = Pk / sum(Pk)
    prob += w * Pk

if sum(prob) > 0:
    prob = prob / sum(prob)

best_letter = argmax_over_unseen(prob)
```

15 API Usage Walkthrough

1. `start_game(practice=True, verbose=True)` resets state and calls `/new_game` to receive `{game_id, word, tries_remains}`.
2. While `tries_remains > 0`:
   (a) `guess_letter = guess(word)` (n-gram pipeline).
   (b) `POST /guess_letter` with `{game_id, letter}`.
   (c) If status is `success/failed`, stop; else update `word` and `tries_remains`.

16 Glossary

- **Mask**: partially revealed word with `_` for unknowns.
- **n-gram**: contiguous sequence of n characters; we store frequency tables for $n = 1..5$.
- **Blend/Interpolation**: weighted sum of per-level probability vectors.
- **Pruning/Re-calibration**: removing dictionary words that contain letters proven incorrect so far.