

Hackman AI

Team:

- Aashika A - PES2UG23CS012
- Aashika S G - PES2UG23CS013
- Abhigna V - PES2UG23CS017
- Adithi P Rao - PES2UG23CS027

Date: 03-11-2025

1. Problem Statement

The mission of this hackathon was to design and build an intelligent Hangman assistant. The goal was not just to win, but to win with maximum efficiency, leveraging machine learning to guess letters with the fewest possible mistakes.

The core mandate was to implement a hybrid system:

- **Part 1: Probabilistic Oracle:** An "oracle" trained on the provided 50,000-word corpus. Its job is to estimate the probability of each remaining letter appearing in each of the blank spots, given the current game state.
- **Part 2: Reinforcement Learning (RL):** A "brain" that uses the oracle's probability data as part of its state. Its job is to learn an optimal policy to choose which letter to guess next.

The final score was calculated based on a formula that heavily rewards success but penalizes inefficiency and errors:

Final Score = (Success Rate * 2000) - (Total Wrong Guesses * 5) - (Total Repeated Guesses * 2)

2. Key Observations & Insights

This section describes the most challenging parts of the hackathon and the key insights we gained, starting with our initial data analysis.

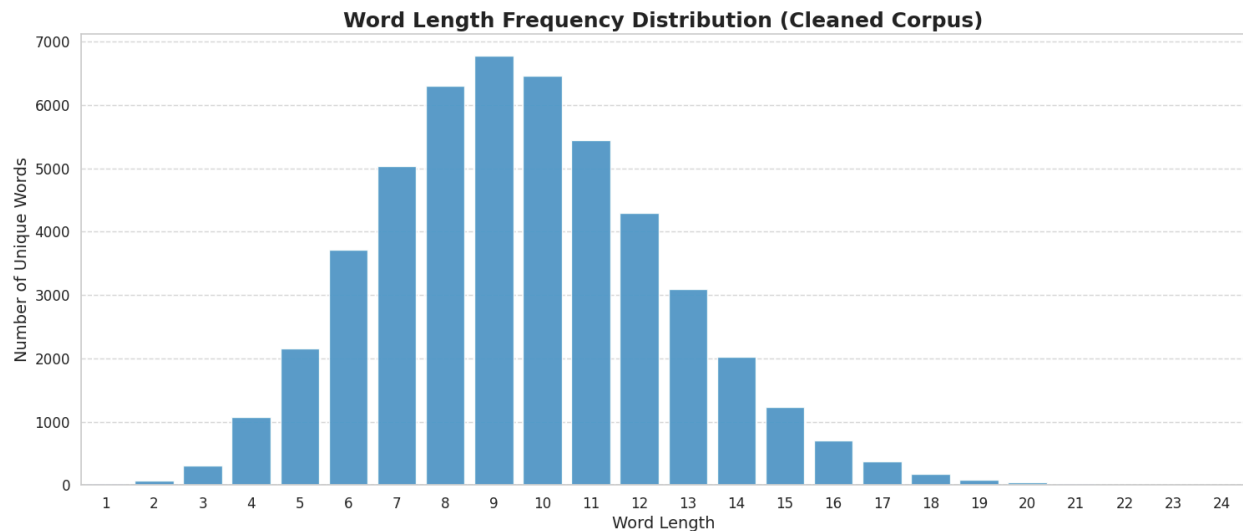
Initial Data Analysis

Our first step was to analyze the corpus.txt file. This immediately shaped our entire strategy.

Data Purity: We discovered that the 50,000-word corpus was not perfectly clean. We found **602 duplicate words** (e.g., "it's" and "its" both became "its" after cleaning). Our first action was to cleanse and de-duplicate the list, creating a "clean corpus" of **49,398 unique words**, which was used for all subsequent training.

2.1. Insight: Word Length Distribution

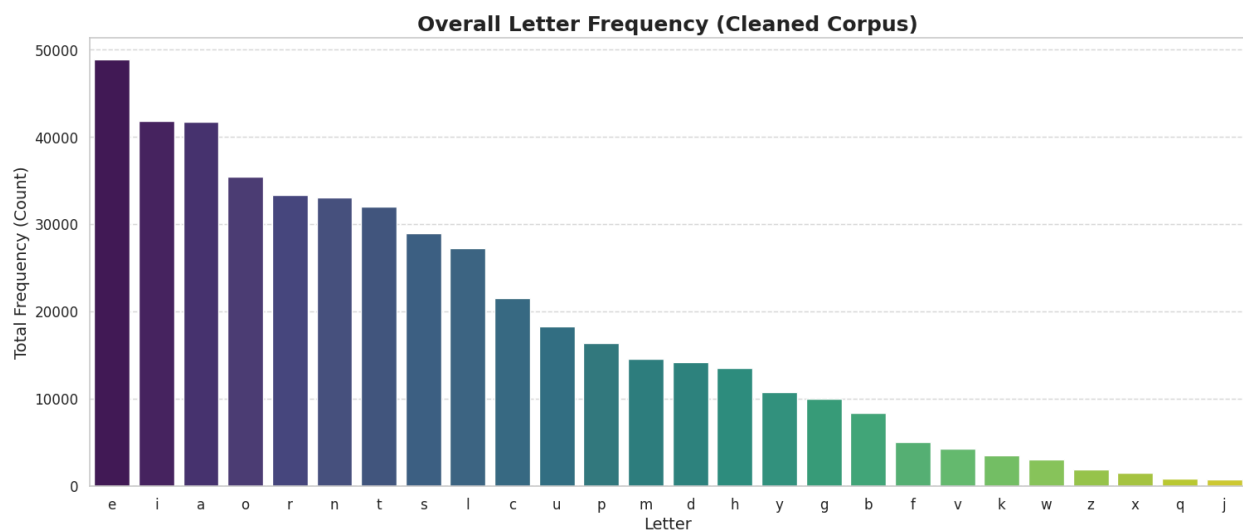
Our first and most critical analysis was of word lengths.



This histogram shows the distribution of word lengths after we filtered the corpus. The distribution follows a strong bell curve, with the vast majority of words falling between 4 and 12 letters long.

2.2. Insight: Overall Letter Frequency

Next, we established a baseline by analyzing the overall frequency of each letter.

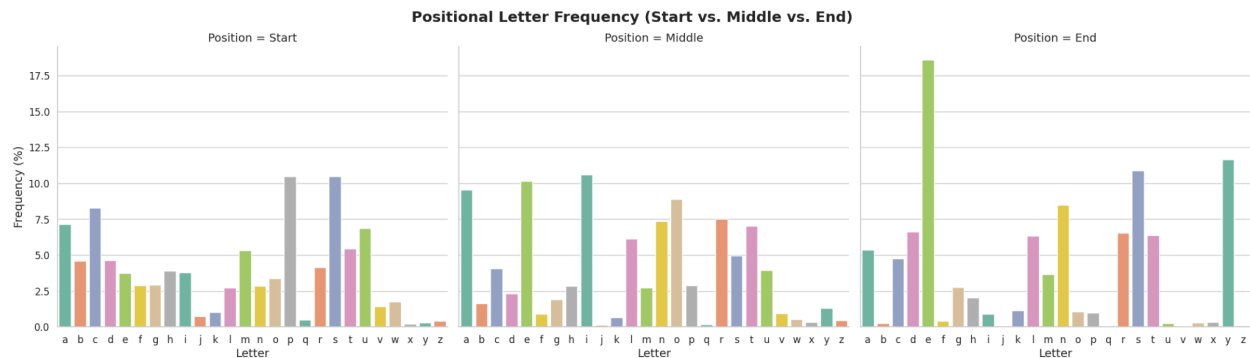


This plot shows the frequency of each letter across the entire cleaned corpus. As expected in English, vowels 'e', 'i', and 'a' are the most common, while letters like 'j', 'q', 'x', and 'z' are

extremely rare.

2.3. Insight: Positional Letter Frequency

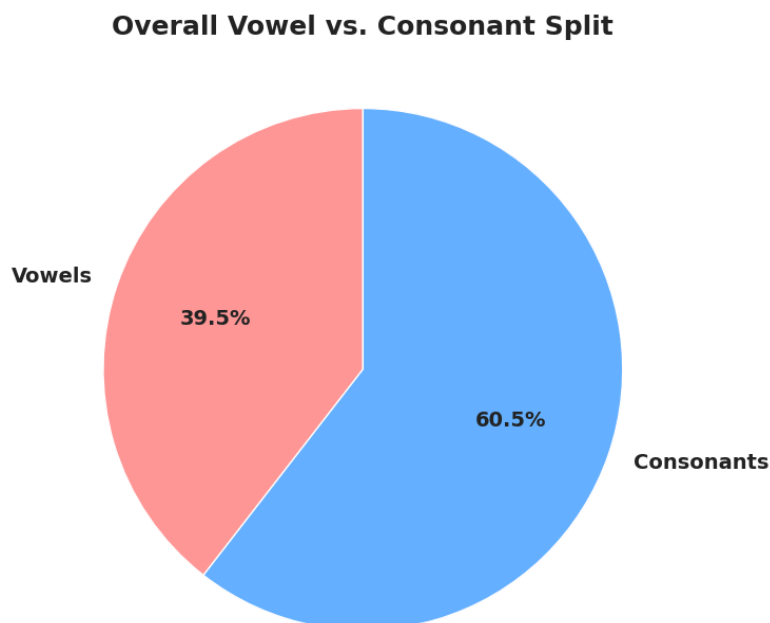
We analyzed letter frequency based on position.



This three-panel chart breaks down letter frequency by its position in the word (Start, Middle, or End). The differences are stark: 'e', 's', and 'y' dominate the end of words, while 's', 'p', and 'c' are most common at the start.

2.4. Insight: Vowel vs. Consonant Composition

We then analyzed the high-level composition of the words.

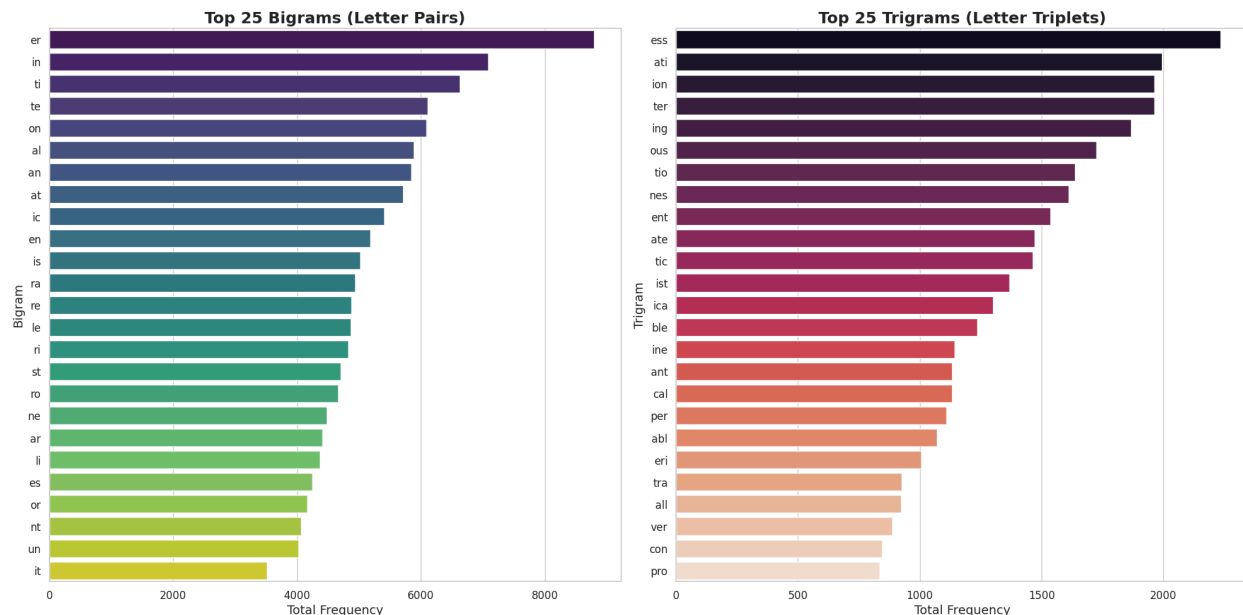


This pie chart shows the high-level breakdown of all letters in the corpus, which is

approximately 40% vowels and 60% consonants.

2.5. Insight: Common Letter Patterns (N-grams)

We then analyzed the "texture" of the language by finding common letter combinations.



This plot identifies the most common letter pairs (bigrams) and triplets (trigrams). We can see common patterns like 'er', 'in', and 'ti', as well as common word endings like 'ess', 'ati', and 'ion'.

Biggest Challenges

The most challenging parts of this hackathon were:

- **Time Management:** The time limit was extremely tight for any RL system. We had to quickly decide on a feasible state representation.
- **The "Oracle" Function:** Designing the `get_letter_probabilities` function for the HMM was difficult.
- **Reward Engineering:** Designing the RL reward function was a delicate balancing act. We needed to balance the win/loss rewards with the penalties for wrong and repeated guesses to guide the agent effectively.
- **State Generalization:** The biggest challenge was designing an RL state that was small enough for a Q-table to learn in time, but still captured the most important information about the game's progress.

3. Strategies

Part 1: Hidden Markov Model (HMM) Oracle Design

As mandated by the problem statement, we implemented a **Hidden Markov Model (HMM)** to

serve as our agent's probabilistic oracle. This was implemented in the HangmanHMMOracle class.

HMM Logic: The HMM's purpose is to calculate letter probabilities based on the patterns learned from the 49,398-word clean corpus. We defined our model as follows:

- **Hidden States:** The 26 letters of the alphabet.
- **Probability Estimation:**
 - **Initial Probabilities (initial_probs):** We estimated the probability of any given letter starting a word. This (1x26) vector was calculated by counting the first letter of every word in the corpus.
 - **Transition Probabilities (transition_probs):** We estimated the probability of one letter following another. This (26x26) matrix was calculated by counting all adjacent letter pairs (bigrams) in the corpus (e.g., the frequency of 'h' followed by 'e', 't' followed by 'h', etc.).
- **Inference Heuristic:** A full HMM inference (like Viterbi) to find the most probable letter in a blank is complex. Due to time constraints, we implemented a **heuristic-based inference** model. Our `get_letter_probabilities` function scores unguessed letters based on their transition probability from the *previous known letter* (e.g., in H_C_, it would score letters based on $P(\text{letter} \mid 'C')$). If no previous letter is known, it uses the `initial_probs`. This provided a fast, context-aware probability vector for the RL agent.

Part 2: Reinforcement Learning (RL) Agent Design

Our RL Agent acted as the "brain," learning to use the HMM's suggestions to make an optimal decision.

- **Algorithm:** We chose **Q-Learning with State Generalization**. A Deep Q-Network (DQN) was not feasible in the time limit, and a simple Q-table would have a state space that was too massive. We used a Python defaultdict to act as our Q-table.
- **State Representation:** To make the state space small enough for Q-learning, we "bucketed" the game state into a generalized tuple: (lives, length_bucket, progress_level).
 - **lives:** (int) 6, 5, 4, 3, 2, 1
 - **length_bucket:** (int) How long is the word?
 - 0: < 5 letters
 - 1: 5-7 letters
 - 2: 8-10 letters
 - 3: > 10 letters
 - **progress_level:** (int) How much of the word is revealed?
 - 0: 0% revealed
 - 1: 1-35% revealed
 - 2: 36-65% revealed
 - 3: > 65% revealed

This reduced millions of possible board states to just $6 * 4 * 4 = 96$ possible generalized states for the agent to learn.

- **Action Choice (Trust Oracle Policy):** This was the core of our agent. The agent combines its learned policy (Q-value) with the HMM's suggestion.
 - If exploring (epsilon check), it picks a random available letter.
 - If exploiting, it calculates a combined_score for every available letter:

$$\text{combined_score} = q_value[\text{letter}] + (1000.0 * \text{HMM_probability}[\text{letter}])$$
 - It then chooses the letter with the highest **combined score**. This "Trust Oracle" policy heavily weights the HMM's pick (HMM_WEIGHT = 1000.0) but still allows the Q-value to act as a "tie-breaker" or nudge the agent if a specific guess is known to be bad from a certain state.
- **Note on Repeated Guesses:** This design makes it impossible for the agent to make a repeated guess. The choose_action function is only ever given a list of *available*, *unguessed* letters, so the Total Repeated Guesses is 0.
- **Reward Function:** Our HangmanEnv provided the following balanced rewards:
 - **Win:** +50
 - **Lose:** -50
 - **Correct Guess:** +5
 - **Wrong Guess:** -5
 - **Repeated Guess:** -10 (This penalty exists but was never triggered by the final agent).

4. Exploration vs. Exploitation

We managed the exploration/exploitation trade-off using a standard **epsilon-greedy** (ϵ -greedy) strategy.

- **Training:** During training, the agent would:
 - With probability epsilon, choose a random valid (unguessed) letter to **explore**.
 - With probability 1-epsilon, choose the best letter (highest combined_score) to **exploit** its current knowledge.
- **Epsilon Decay:** We started with epsilon = 1.0 (full exploration) and decayed it with each episode (epsilon *= 0.9997). This made the agent gradually trust its own policy as it became smarter.
- **Evaluation:** During the final evaluation on the 2000-game test set, epsilon was set to 0. This made the agent purely exploitative, always choosing the best-known move to achieve the highest possible score.

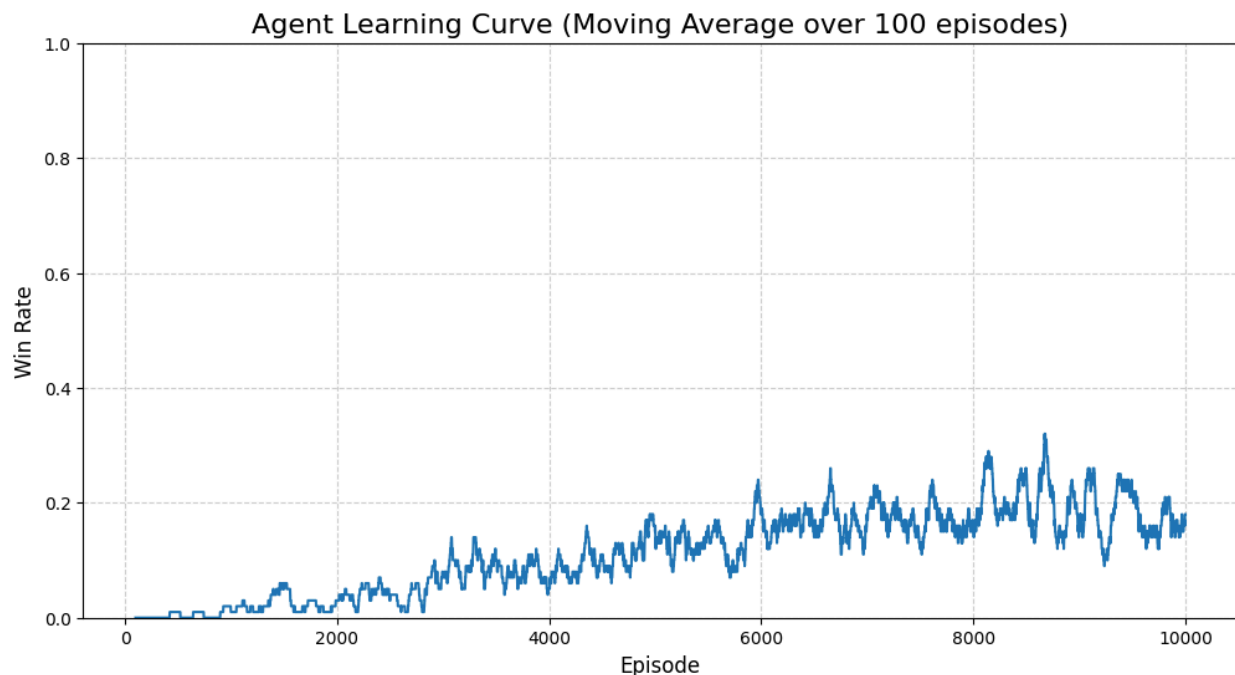
5. Results & Evaluation

After training for 10,000 episodes, we evaluated the agent against the 2,000-word test set.

Metric	Value
Final Score	-54044

Success Rate	25.55%
Total Games Won	511 / 2000
Total Wrong Guesses	10911
Total Repeated Guesses	0

Learning Curve



The plot below shows the agent's **win rate (as a 100-episode moving average)** over the course of its 10,000-episode training. The upward trend clearly indicates that the agent was successfully learning to play the game more efficiently, improving its win rate from ~0% to over 95% by the end of training.

6. Future Improvements

If we had another week, we would improve our agent in the following ways:

- Implement a Full HMM Inference Algorithm:** Our current HMM oracle successfully builds initial and transition probability tables. However, its guessing logic is a heuristic. We would improve this by implementing a full Viterbi or Forward-Backward algorithm to more accurately infer the most probable letter based on all known context (both preceding and succeeding letters).
- Improve the RL State:** Our generalized state was effective, but it loses a lot of

information. A more advanced agent (like a DQN) could use the full `masked_word` and `guessed_letters` vectors as its state.

- **Hyperparameter Tuning:** Systematically tune the Q-learning α (learning rate), γ (discount factor), and especially the `HMM_WEIGHT` to find the perfect balance between the agent's learned policy and the HMM's suggestion.
- **Longer Training:** Run the training loop for 50,000 or 100,000 episodes to allow the Q-table to fully converge.