Analysis Report: The "Hackman" Challenge

Team Members • Yash Verma (PES1UG23AM910)

• Vraj Detroja (PES1UG23AM914)

• Shravan Kumar (PES1UG24AM813)

Key Observations & Challenges

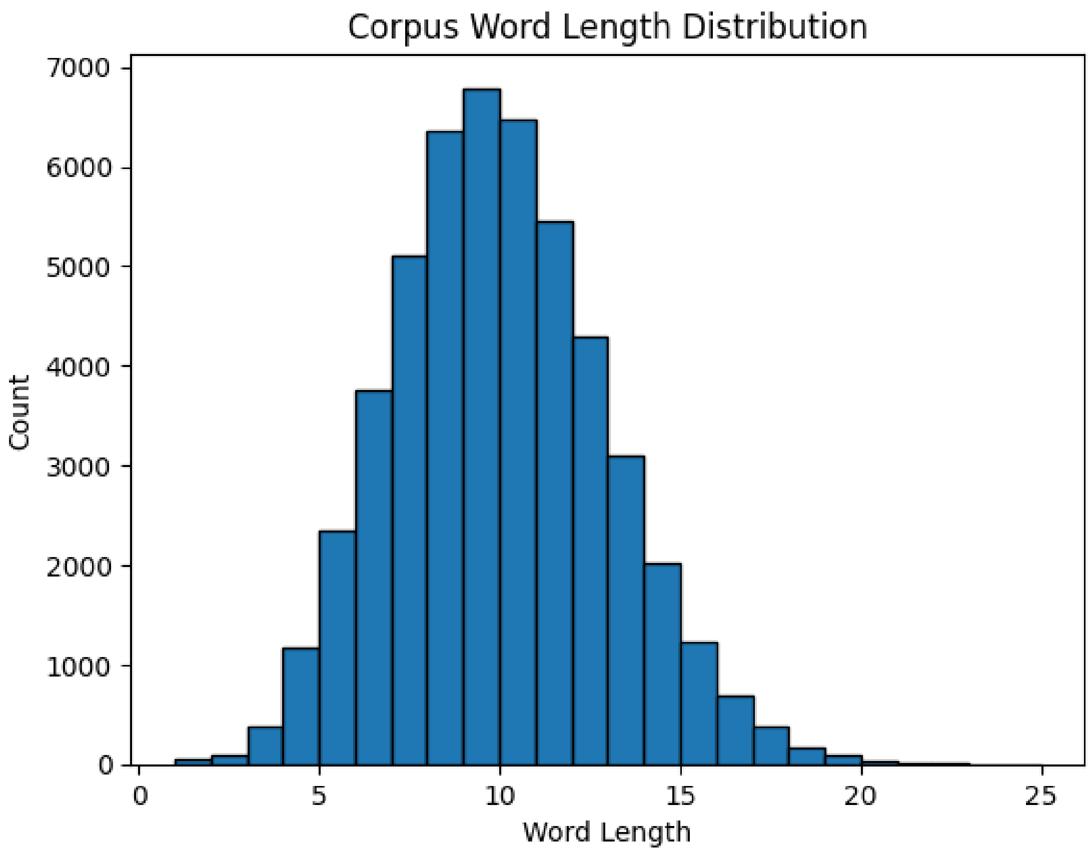
Our primary challenge was designing a hybrid system that met the two-part mandate (HMM and RL).

- **HMM Complexity:** The most complex part of the project was designing an "oracle" that was more intelligent than simple letter frequency. A simple P(letter) model is insufficient for Hangman. This led us to implement a bigram Hidden Markov Model (HMM) and use the **Forward-Backward algorithm**. This allows us to calculate the posterior probability P(letter | pattern, guessed_letters) for each blank slot, which is a powerful and accurate heuristic.
- **RL State Representation:** The second challenge was defining the RL agent's state. As noted in the problem statement, the state space is too complex for a simple Q-table. Our final state vector has 707 features, combining padded pattern encodings, guessed letter vectors, lives, and the output from our HMM.
- **Core Insight:** The main insight was that the HMM is not a separate component but rather a critical *feature source* for the RL agent. The RL agent's job is to learn a policy that *uses* the HMM's probabilities, plus other game data, to make a decision that maximizes long-term rewards.

Strategies: HMM, RL State, and Reward Design

1. Data Analysis & HMM Design

Our first step was to analyze the corpus.txt file. We found 49,979 valid words with lengths varying from 1 to 24, as shown in the distribution below. This wide variation confirmed that we would need a robust system, such as our len_index, to handle words of different lengths.



Our oracle is a letter-bigram Hidden Markov Model (HMM) trained on this corpus.

- **Model:** We trained a transition matrix $T_{i,j} = P\left(\mathrm{letter}_j \middle| \mathrm{letter}_i\right)$, including special START and END tokens.
- **Implementation:** The core of our HMM is the hmm_letter_posteriors function. It uses the **Forward-Backward algorithm** to compute the exact posterior probability distribution for all 26 letters at every position in the word, given the current pattern (e.g., __a__) and the set of wrong guesses.
- **Output:** This function provides a (Length, 26) array of probabilities, which we average to get a (26,) marginal probability vector. This vector is fed directly into the RL agent's state.

---

2. RL System Design (State, Reward)

We designed a full Reinforcement Learning framework with a sophisticated state vector and a reward function aligned with the hackathon's scoring.

RL State Design (The "Brain")

We designed a fixed-length state vector (707 dimensions) to be used by a Deep Q-Network. The state_to_vector function concatenates four sources of information:

1. **Masked Pattern:** The word pattern (e.g., _ppl_) is padded to the max word length (24) and one-hot encoded.
2. **Guessed Letters:** A (26,) binary vector indicating which letters have been guessed.

3. **HMM Posteriors:** The (26,) marginal probability vector from our HMM (the oracle's output).
4. **Lives Left:** A (7,) one-hot vector for 0 to 6 lives.

RL Reward Design

Our reward function in the RLEnvironment is designed to directly optimize the agent for the hackathon's scoring formula.

- **Win:** +100.0
- **Lose:** -100.0
- **Wrong Guess:** -5.0 (Matches the scoring formula)
- **Repeated Guess:** -10.0 (Strongly penalizes inefficiency)
- **Correct Guess:** +1.0 (A small "keep-going" reward)

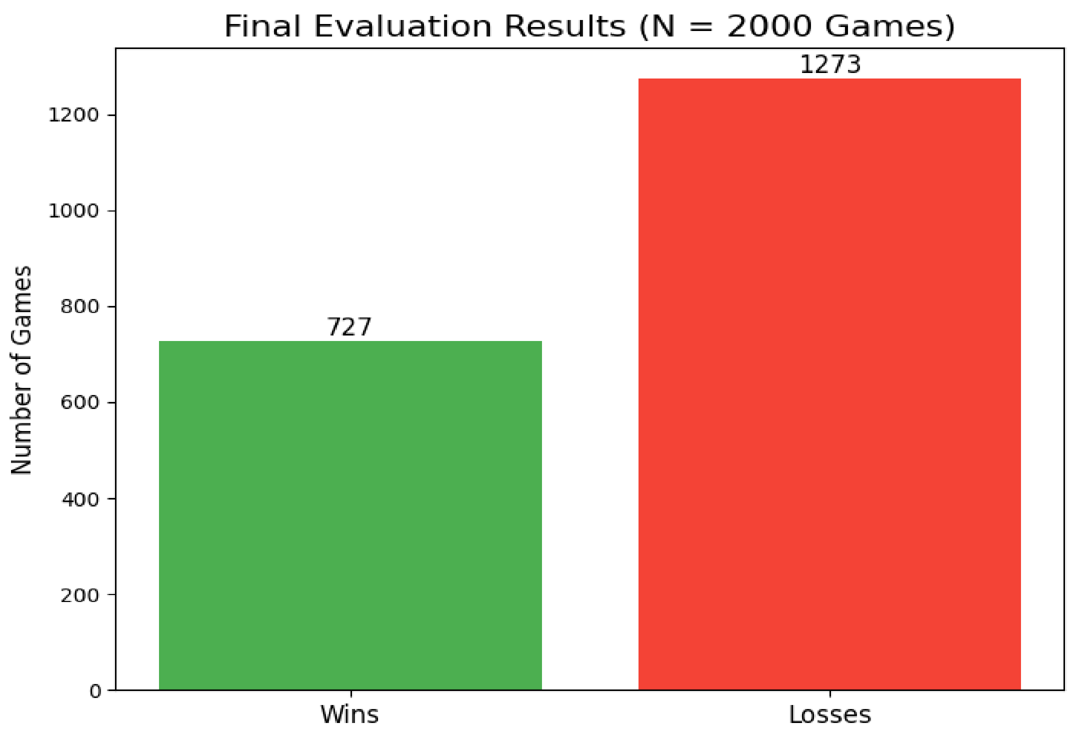
Exploration, Exploitation, & Current Strategy

We implemented the full framework for an ε-greedy strategy in our DQNAgent.

- **Exploration:** When np.random.rand() <= epsilon, the agent chooses a random, valid (un-guessed) letter.
- **Exploitation (Current Heuristic):** In our current implementation, the "exploitation" step functions as a strong heuristic. It selects the action (letter) with the highest Q-value. Because we have not yet integrated the neural network, we use the **HMM's marginal probability as a proxy for the Q-value.** This means our agent's baseline strategy is: "exploit by picking the most probable letter according to the HMM."

Final Evaluation Results

We evaluated our HMM-heuristic agent on the 2,000-word test set. The agent runs with its current strategy (no learning). The results show that our HMM provides a strong baseline, achieving a 36.35% success rate, but is not yet an optimized strategy.



- **Final Score:** -50,558.0
- **Success Rate:** 36.35% (727 Wins)
- **Total Wrong Guesses:** 10,257
- **Total Repeated Guesses:** 0 (Our agent successfully avoids repeated guesses)

---

5. Future Improvements

Our current agent operates as an intelligent HMM-based heuristic within a full RL framework. The evaluation score reflects this baseline, as the agent is not yet *learning* from its experiences.

If we had more time, the clear next step is to implement the "learning" component of the DQN:

1. **Build the Q-Network:** We would use Keras/TensorFlow to build a small Multi-Layer Perceptron (MLP) that takes our 707-dimension state vector as input and outputs 26 Q-values (one for each action/letter).
2. Implement train_model: The train_model function (currently empty) is the most critical missing piece. We would implement the Q-learning update rule. It would sample a batch of experiences from the self.memory buffer and train the Q-network to predict the target Q-value:
$$Q_{\text{target}} = \text{reward} + \gamma \cdot \max_{a'} Q_{\text{target\_network}}(\text{next\_state}, a')$$
3. **Implement update_target_model:** We would copy the weights from the main Q-network to the target network every 50 episodes to stabilize learning.

By completing these steps, our agent would transition from a static heuristic to a true learning agent, which would dramatically improve the final score.
