PES UNIVERSITY

UE23CS352A – MACHINE LEARNING
LAB REPORT

ON

"Hackman: Hybrid AI agent"

SUBMITTED BY

NAME	SRN
1) A R Keerthana	PES2UG23CS001
2) Amrutha P J	PES2UG23CS059
3) Ananya A C	PES2UG23CS061
4) Ananya Lakshmi	PES2UG23CS062

AUGUST – DECEMBER 2025

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

ELECTRONIC CITY CAMPUS,

BENGALURU – 560100, KARNATAKA, INDIA

Analysis Report:

The solution successfully implements the **hybrid system** mandated by the challenge, combining a probabilistic model (Part 1) with a Reinforcement Learning agent (Part 2) to create an intelligent Hangman assistant.

1. The Dataset & Probabilistic Model (HMM):

The problem requires the agent's "intuition" to come from a probabilistic model, described as a **Hidden Markov Model (HMM)**, which must be trained only on the provided corpus.txt file.

Implementation: Our approach fulfills this requirement not with a formal HMM library, but by building a custom, highly effective probabilistic oracle trained on the corpus.

- **Corpus Training:** The load_and_clean_corpus function reads corpus.txt, cleaning and grouping all 49,975 valid words by their length (words_by_length). This grouping strategy is a key design choice that directly addresses the "HMM Complexity" hint regarding words of different lengths.
- **N-gram Models:** The train_ngram_models function is run on the entire word list to calculate unigram, bigram, and trigram probabilities. This serves as a contextual model.
- **The Oracle Function:** The core of Part 1 is the get_letter_probabilities function. This function acts as the required oracle by estimating the probability of each letter for the blank spots. It operates in two modes:
    1. **High-Confidence (Regex Match):** It first generates a regex pattern for the current state (e.g., _PPL_) and attempts to find all matching words within the pre-filtered word list for that specific length. If matches are found, it calculates a precise probability distribution from the letters in the blank spots and returns a **confidence score of 1.0**.
    2. **Low-Confidence (Hybrid Fallback):** If no words in the corpus match the current pattern, the system falls back to the get_hybrid_fallback_probs function. This function provides a probability distribution by combining the contextual N-gram model (70% weight) with general letter frequencies for that specific word length (30% weight). In this case, it returns a **confidence score of 0.0**.

This two-mode system (Regex + N-gram) serves as the "probability distribution over the alphabet" and is the crucial piece of information fed to the RL agent, as mandated.

2. The Reinforcement Learning (RL) Agent

The problem requires an RL "brain" that uses the HMM's information to choose the optimal letter. This requires defining a custom environment, state, action, and reward.

Implementation:

- **Environment** : The class HangmanEnvironment was built as required.
- **Reward Function** : The reward function is designed to directly optimize for the final score formula. The penalties for wrong and repeated guesses are identical to the formula:
    1. reward_wrong_guess = -5
    2. reward_repeated_guess = -2
    3. Positive rewards (reward_win = 100, reward_correct_guess = 5) are used to guide the agent toward winning.
- **RL Algorithm** : Our approach uses a QLearningAgent, following the hint to "start simple" with a Q-learning table.
- **State Representation:** The **state** is the most critical part of this hybrid system. It is defined in _get_state_and_probs as a tuple: (current_lives, num_blanks, int(confidence)).
    1. This state representation is compact and highly effective. It includes lives left , a summary of the masked word (num_blanks), and most importantly, the **confidence score from the HMM oracle**.
    2. This confidence bit explicitly links the RL agent's "brain" to the HMM's "intuition", fulfilling the core hybrid mandate.
- **Action Space:** Instead of a complex 26-letter action space, the agent learns from a simplified 3-action policy:
    1. **Action 0:** Guess the HMM's **#1 most probable letter**.
    2. **Action 1:** Guess the HMM's **#2 most probable letter**.
    3. **Action 2:** Ignore the HMM and guess from a **generic fallback list** (GENERIC_LETTERS). This is implemented in the _map_action_to_letter function. The agent's entire job is to learn which action (which "advice") to take given the state (lives, blanks, and HMM confidence).

3. Training & Exploration

The agent must be trained in the environment using an exploration strategy.

Implementation:

- **Training Loop:** The def train function serves as the complete training loop. The agent was trained for **75,000 episodes**.
- **Exploration:** The QLearningAgent uses an **epsilon-greedy** strategy, as hinted in the problem statement. In choose_action, the agent picks a random action if random.uniform(0, 1) < self.epsilon. The training logs show Epsilon starting at 1.0 and **decaying** over time to a minimum of 0.01, allowing the agent to shift from exploration to exploitation.

4. Evaluation & Final Score

The agent's performance must be evaluated by playing 2000 games and scored using the final score formula.

Implementation:

- **Evaluation :** The evaluate_agent_on_test_file function was run on the test.txt file, which contains 2000 words. The agent's epsilon was set to 0.0 for pure exploitation (no random guesses).
- **Plots :** As required, the notebook generated and saved plots (reward_plots_confidence_strategy.png) showing the "Agent's Learning Curve" for both "Moving Avg. Reward" and "Moving Avg. Success Rate %", which clearly trend upward during training.
- **Final Results :** The evaluation output provides all required metrics:
  - **Total Games Played :** 2000
  - **Success Rate :** 32.25% (645 / 2000)
  - **Total Wrong Guesses :** 10445
  - **Total Repeated Guesses :** 0
  - **Final Score : -51,580.00**
    - Calculation*:* (Success Rate * 2000) - (Total Wrong Guesses * 5) - (Total Repeated Guesses * 2)
    - (0.3225 * 2000) - (10445 * 5) - (0 * 2) = 645 - 52225 - 0 = -51580.00

This solution successfully builds the complete hybrid system, trains it exclusively on the provided corpus, and evaluates it on test.txt according to all problem statement criteria.

*****