Department of Computer Science Engineering

UE23CS352A: Machine Learning Hackathon

Hackman

1. The Challenge

Hangman is a classic game of incomplete information and probabilistic reasoning. While humans often rely on intuition and simple letter frequency (like guessing 'E,' 'T,' 'A,' or 'O'), a truly intelligent agent can do better. It can learn the *contextual* patterns of a language to make strategic, informed decisions.

Your challenge is to build such an agent.

Your mission is to design and build an intelligent Hangman assistant that effectively guesses letters to solve puzzles with maximum efficiency. The goal is not just to win, but to win with the fewest possible mistakes, leveraging machine learning to peer into the hidden word.

2. The Mandate

To solve this, you are required to implement a hybrid system that combines probabilistic modeling with decision-making:

Part 1: Hidden Markov Model

Your agent's "intuition" will come from a **Hidden Markov Model (HMM)**. You must train an HMM on the provided 50,000-word corpus.

- **Purpose:** The HMM's job is to act as an oracle. Given the current state of the game, it should be able to estimate the probability of each *remaining* letter appearing in each of the blank spots.
- **Implementation:** You must decide on the HMM's structure.
    - What are the hidden states?
    - What are the emissions?
- **Output:** The HMM will provide a probability distribution over the alphabet, which will serve as a crucial piece of information for your decision-making agent.

Part 2: Reinforcement Learning

Your agent's "brain" will be a **Reinforcement Learning (RL) agent**. This agent's job is to use the information from the HMM to choose the *optimal* letter to guess next.

- **Environment:** You will need to build a Hangman game environment that your agent can interact with.
- **Agent:** This is the core of your RL problem. You must define:
    - **State:** What information does your agent need to make a decision? (e.g., the current masked word, the set of guessed letters, the number of lives left, the probability distribution from the HMM).
    - **Actions:** The set of possible actions is to guess any letter from the alphabet that has not already been guessed.
    - **Reward:** You must design a reward function to guide your agent. This is critical. You want to:
        - **Maximize:** Success rate (winning the game).
        - **Minimize:** Wrong guesses (losing a life) and repeated guesses (inefficiency).

3. The Dataset

- **Corpus:** You will be provided with a single corpus.txt file containing 50,000 English words.
- **Rules:** This corpus is your **only** source of truth. Your HMM must be trained on it, and your RL agent will be trained and evaluated *only* on words from this list. You may not use external word lists or pre-trained language models.

4. Scoring & Evaluation

Your agent will be evaluated by playing 2000 games of Hangman (with 6 wrong guesses allowed per game) against a **hidden test set** of words which is given to you.

Your final score will be calculated based on the following formula:

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

- **Success Rate:** The percentage of the 1,000 games your agent wins.
- **Total Wrong Guesses:** The sum of all incorrect guesses across all 1,000 games.
- **Total Repeated Guesses:** The sum of all repeated guesses (guessing a letter that was already guessed).

This formula heavily rewards success but penalizes inefficient and incorrect guesses. A high score demands both accuracy and intelligence.

5. Technical Guidance & Hints

- **HMM Complexity:** How will you handle words of different lengths? You might train separate HMMs for different lengths or use padding/special tokens.
- **State Representation:** A simple RL state might be a string _PPL_:[ESR]. A more complex one might be a vector combining a one-hot encoding of the masked word, a binary vector of guessed letters, and the probability vector from your HMM.
- **RL Algorithm:** Start simple. A basic Q-learning table might work if your state space is small. If your state representation is complex, you may need to explore a Deep Q-Network (DQN) where a neural network learns to approximate the Q-value function.
- **Exploration:** Your agent needs to explore. Think about your exploration-exploitation strategy (e.g., ε-greedy, where ε decays over time) to ensure your agent learns effectively.

6. Deliverables

1. Viva
2. Demo of your solution

You must also submit the following:

1. **Your Python notebooks including the following:**

The construction, training, etc. of your Hidden Markov Model on the corpus.txt file.

Detailing your RL environment, agent design (Q-learning, DQN, etc.), state/action/reward definitions, and the complete training loop.

Evaluation results including: Final Score, plots of your agent's learning (e.g., reward per episode), Final results: Success Rate, Avg. Wrong Guesses, Avg. Repeated Guesses.

2. **Analysis_Report.pdf**: A file answering the following:
   - **Key Observations:** What were the most challenging parts? What insights did you gain?
   - **Strategies:** Discuss your HMM design choices. Detail your RL state and reward design and why you chose them.
   - **Exploration:** How did you manage the exploration vs. exploitation trade-off?
   - **Future Improvements:** If you had another week, what would you do to improve your agent?

Good luck!