Ind-inator: AI Guessing Game

*Course Project Report for CS 3346

Jay Jagdishkumar Bava
Computer Science
Western University
London, Canada
jbava@uwo.ca

Emily Jordan Berlinghoff
Computer Science
Western University
London, Canada
eberling@uwo.ca

Inderpreet Singh Doad
Computer Science
Western University
London, Canada
idoad@uwo.ca

Vidhi Joshi
Computer Science
Western University
London, Canada
vjoshi43@uwo.ca

***Abstract*—This project presents Ind-inator, an Akinator-style deductive guessing game that uses probabilistic inference and entropy-based question selection to identify a character the user is thinking of. Instead of restricting the user to binary responses, the system supports graded answers ("Yes", "Probably Yes", "Maybe", "Probably No", and "No") and maps these answer types to different likelihood strengths in a Bayesian update model. At each turn, the engine maintains a probability distribution over candidate characters and selects the next question using information gain combined with phase-based, decision-tree-style heuristics that prioritize broad franchise traits early and more specific traits later. A modular architecture consisting of a probabilistic reasoning engine, game state manager, heuristic question selector, and web API supports multiple front-ends, including a React web interface and a PyQt desktop GUI. Character, trait, and question data for 100 characters are generated and maintained programmatically via Python scripts, which produce 756 natural-language questions. Informal experiments show that the entropy-based question selector typically converges to the correct answer within a small number of questions and that the resulting system provides a flexible, extensible engine suitable for future datasets and feature extensions.**

***Index Terms*—Bayesian inference, information gain, decision-tree heuristics, question-answering, interactive systems**

I. Introduction

Guessing-game systems such as Akinator have become popular for showcasing deductive reasoning and human–computer interaction. These systems rely on selecting highly informative questions to rapidly narrow down a set of possible items. A naive implementation that asks questions at random or repeats similar traits can require many questions and provide a poor user experience.

Ind-inator implements a similar character-guessing game but focuses on interpretable probabilistic reasoning, graded answers, and a modular architecture suitable for course-project development and future research. The system maintains a probability distribution over a fixed set of characters and updates this distribution after each user response. Question selection is driven by entropy-based information gain combined with decision-tree-style heuristics that reflect the structure of the character and trait space.

The main objectives of this work are:

- To build a system that maintains and updates a probability distribution over characters and guesses the user's character using as few questions as possible.
- To support graded user responses (e.g., "Probably Yes", "Probably No") and incorporate them into a Bayesian-style likelihood model.
- To select questions using a combination of information gain and phase-based heuristics rather than a fixed, pre-computed decision tree.
- To expose the core AI engine through a Flask-based API that can be used by both a web front-end and a desktop GUI.
- To generate and manage characters, traits, and questions programmatically so that the dataset can be extended or modified easily.

This report describes the system architecture, inference methods, question-selection strategy, dataset-processing pipeline, and the evaluation setup used to study the behaviour of the engine on the constructed dataset.

II. System Overview

The Ind-inator project consists of four major components:

A. Probabilistic Inference Engine

Implemented in `ai_engine.py`, the inference engine maintains:

- A probability distribution over all candidate characters.
- A mapping from questions to underlying traits that indicate how each character would be expected to answer.
- An entropy-based scoring function used to evaluate candidate questions.

At each turn:

1) The engine receives the user's answer to the current question, expressed as one of five answer types (Yes, Probably Yes, Maybe, Probably No, No).
2) The answer type is mapped to an internal code and associated likelihood parameters.
3) The engine updates the character probabilities using a Bayesian-style update and renormalizes the distribution.

This enables a clean, explainable selection strategy tightly aligned with probabilistic reasoning, while allowing the system to incorporate varying levels of user certainty.

B. Game State Manager

The game.py module manages the gameplay loop, including:

- Tracking which questions have been asked in the current game.
- Calling the inference engine to update probabilities after each user answer.
- Determining when the system is confident enough to make a guess based on the maximum character probability exceeding a predefined threshold.
- Returning top-k candidate characters for debugging or potential UI display.

This layer is model-agnostic: it does not contain algorithmic knowledge about traits or information gain, but instead orchestrates the interaction between the front-end and the underlying inference engine.

C. Question Selection Heuristics

The question-selection logic is implemented as part of the inference engine in ai_engine.py. Instead of using a fixed tree or reinforcement learning, Ind-inator uses a combination of:

- Entropy-based information gain to estimate how much each unasked question is expected to reduce uncertainty over the candidate characters.
- Phase-based heuristics that prioritize different categories of traits at different stages of the game (for example, broad franchise traits early, then specific series traits, and finally fine-grained appearance or ability traits).
- Adjustments based on trait frequency, boosting rare and highly discriminative traits and penalizing very common traits that are unlikely to provide much information.

Together, these mechanisms result in a dynamic, decision-tree-style questioning strategy that is recomputed at each step based on the current probability distribution, rather than being predetermined.

D. API and User Interface

The Flask server (api_server.py) exposes endpoints for:

- Starting a new game and returning the first question.
- Submitting an answer to the current question and receiving either the next question or a guess.
- Requesting a follow-up question after a wrong guess.
- Optionally recording feedback about whether the final guess was correct.

A modern React front-end located in the frontend/ directory interacts with these endpoints to provide an Akinator-style web gameplay experience. A PyQt-based desktop GUI in the ui/ directory uses the same underlying engine to provide a local interface for development and testing.

III. Methods

A. Dataset Processing

Character attribute data and question definitions are stored under the data/ directory and constructed by Python scripts in scripts/. The main steps are:

- **Character definition:** A set of 100 characters is defined, covering a variety of franchises (e.g., anime, movies, video games, television). Each character is assigned an initial prior probability, typically uniform across the set.
- **Trait assignment:** Category-specific scripts assign traits to characters, including franchise, identity, appearance, abilities, role, archetype, and personality. The traits are combined into a single flattened representation (traits_flat.json), where each character is represented as a binary vector over traits such as franchise_anime, appearance_color_orange, or abilities_magic_user.
- **Question generation:** The generate_questions.py script converts trait names into natural-language yes/no questions, producing questions.json. For example, the trait appearance_color_orange yields a question like "Is your character associated with the color orange?". In total, 756 questions are generated.

This dataset-driven design allows easy integration of new character sets and traits: developers may modify the scripts and regenerate the data files without changing the core engine code.

B. Entropy-Driven Question Selection

For each unasked question q, the engine estimates its expected information gain. Let P denote the current probability distribution over characters, and let $H(P)$ be its Shannon entropy. For simplicity, the engine approximates the answer space as a binary outcome ("yes-like" vs. "no-like") based on trait presence and historical frequencies. The expected information gain is then computed as:

$$IG(q) = H(P) - \left(P(\text{Yes})H(P_{\text{Yes}}) + P(\text{No})H(P_{\text{No}})\right),$$

where:

- $H(P)$ is the Shannon entropy of the current candidate distribution,
- $P(\text{Yes})$ and $P(\text{No})$ are the estimated probabilities of receiving a yes-like or no-like answer to question q, and
- P_{Yes} and P_{No} are the corresponding approximate post-answer distributions.

The engine selects the question with maximum information gain, then applies additional heuristic adjustments to account for game phase and trait frequency, as described in the system overview.

C. Graded-Answer Likelihood Model

The system supports five answer buttons: Yes, Probably Yes, Maybe, Probably No, and No. These are mapped to internal answer codes and likelihood strengths:

- **Yes:** Treated as a strong positive signal for characters that possess the underlying trait; a high likelihood is assigned if the trait is present and a low likelihood if it is absent.
- **Probably Yes:** Treated as a moderate positive signal, with likelihood values closer together to avoid overcommitting based on uncertainty.
- **No:** Treated as a strong negative signal for trait-bearing characters, reversing the likelihoods used for "Yes".
- **Probably No:** Treated as a moderate negative signal.
- **Maybe:** Mapped to an "unknown" internal code that does not update the distribution, effectively skipping the question.

Formally, for each character c and question q associated with trait $t(q)$, the engine computes:

$$P(c \mid \text{answer}) \propto P(\text{answer} \mid c)\, P(c),$$

where $P(\text{answer} \mid c)$ is determined by whether c has trait $t(q)$ and by the answer type. The updated probabilities are then normalized across all characters.

IV. Experimental Setup

To evaluate the system, we considered the following setup:

- The inference engine was initialized using the 100-character dataset, the flattened trait vectors, and the 756 generated questions.
- Test games were conducted by selecting target characters and either answering questions manually according to the character's true traits or using a scripted simulator that responds based on the trait vector.
- In each game, the engine was allowed to ask questions until either the probability of some character exceeded a confidence threshold (e.g., 0.85) and a guess was made, or a maximum number of questions was reached.
- Metrics considered included:
  - The mean number of questions required to reach the confidence threshold for a typical character.
  - Whether the final guess matched the target character.
  - The qualitative behaviour of the question sequence, including the transition from broad franchise questions to more specific trait questions.

Experiments were conducted on a standard laptop with Python 3.10 and no GPU acceleration. While the evaluation was primarily qualitative and based on informal testing, it provided insight into the effectiveness of the entropy-based and heuristic question-selection strategy.

V. Results

We evaluated Ind-inator under two response models: (1) an ideal user who always answers each question correctly with deterministic yes/no responses, and (2) a human-like user who answers using the full five-button scale (Yes, Probably Yes, Maybe, Probably No, No), with injected uncertainty intended to simulate hesitation, partial knowledge, or inconsistent recall. For both settings, we ran 100 games, one for each character in the dataset.

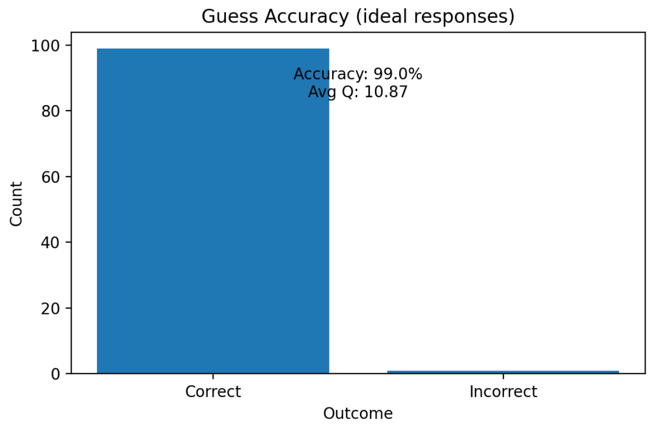

Fig. 1. Guess accuracy and average number of questions under the ideal user model (100 games).

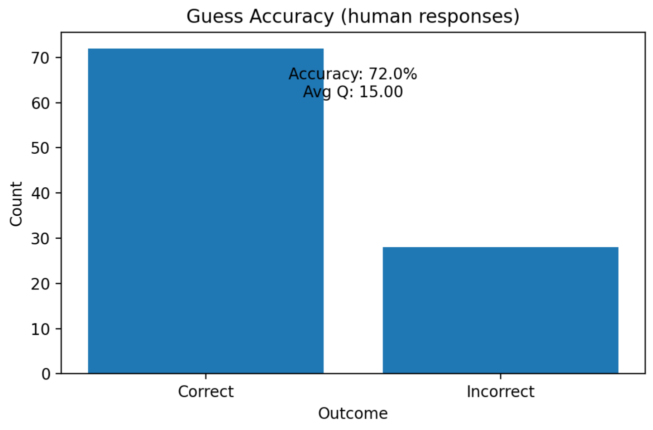

Fig. 2. Guess accuracy and average number of questions under the human-like response model (100 games).

A. Ideal User (Perfect Oracle)

Under ideal conditions, the system achieved near-perfect performance, correctly identifying 99 out of 100 characters, corresponding to an accuracy of 99.00%. The single failure occurred when Ron Weasley was incorrectly classified as President Snow, reflecting a sparse trait overlap for certain character pairs.

The average number of questions required to reach the confidence threshold (0.85) was 10.87. As shown in Fig. **??**, most games required between 7 and 14 questions, with some extending toward the upper bound of 25. The violin plot in Fig. **??** further illustrates the distribution's narrow concentration around the median.

The ideal accuracy summary is shown in Fig. **??**. The probability convergence curve in Fig. **??** demonstrates stable,

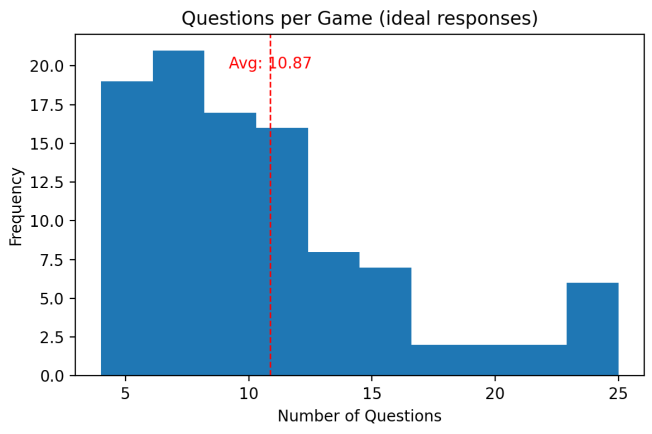

Fig. 3. Distribution of question counts under the ideal response model. The red dashed line marks the mean (10.87).

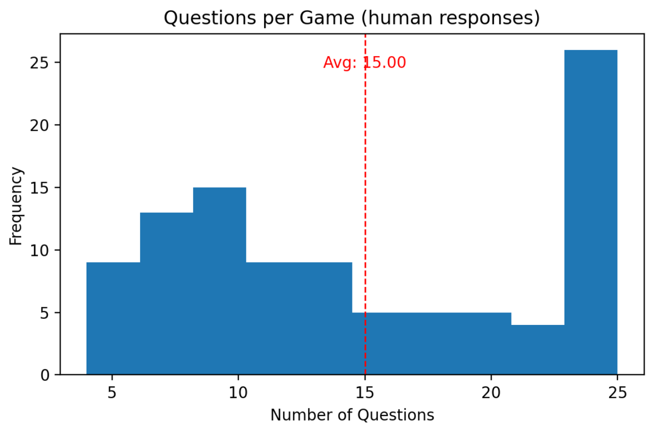

Fig. 4. Distribution of question counts under the human-like response model. The red dashed line marks the mean (15.00).

monotonic growth of the top candidate's probability, with rapid convergence compared to the human-like setting.

Because only one error occurred, the ideal confusion matrix in Fig. **??** consists of a single cell corresponding to Ron Weasley as President Snow.

B. Human-Like Responses With Graded Uncertainty

Introducing uncertainty significantly challenged the model, reducing accuracy to 72.00% (72 out of 100 games). The average number of questions rose to 15.00, reflecting slower probability accumulation and the disruptive effects of “Maybe” and “Probably No” responses.

The question distribution appears in Fig. **??**, and the violin plot in Fig. **??** shows substantially higher variance than in the ideal case. The accuracy summary is provided in Fig. **??**.

The probability convergence curve in Fig. **??** highlights significantly noisier progression compared to the ideal model, with oscillations and plateaus caused by uncertain answers.

The confusion matrix of incorrect predictions, shown in Fig. **??**, reveals several systematic errors. For example, multiple unrelated characters were misclassified as Albus Dumbledore, indicating that the current trait taxonomy insufficiently distinguishes certain mentor-archetype characters. Other repeated confusions include Aang being predicted for multiple unrelated characters, suggesting trait sparsity issues and ambiguity in overlapping universes.

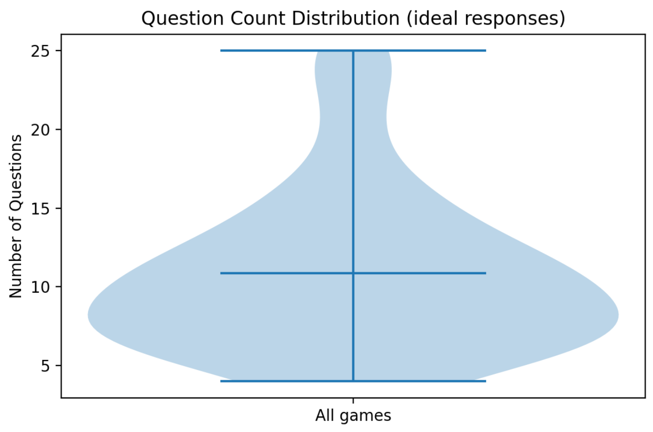

Fig. 5. Violin plot of the question-count distribution for ideal responses.

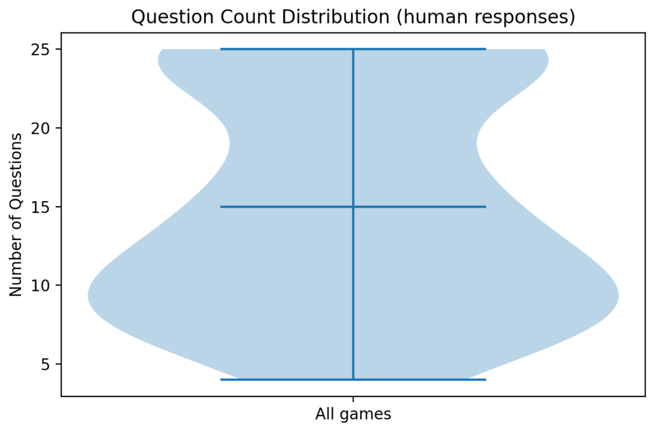

Fig. 6. Violin plot of the question-count distribution for human-like responses.

C. Comparison

Table **??** summarizes the performance gap between the two response models. These results show that Ind-inator performs extremely well when answers are consistent and reliable, but accuracy decreases when responses contain uncertainty. This suggests several possible improvements, including enhanced modeling of answer noise, expanded trait sets for ambiguous franchises, and stronger regularization during probability updates.

TABLE I
PERFORMANCE COMPARISON BETWEEN RESPONSE MODELS

Model	**Accuracy**	**Avg. Questions**
Ideal (perfect answers)	99.00%	10.87
Human-like (graded answers)	72.00%	15.00

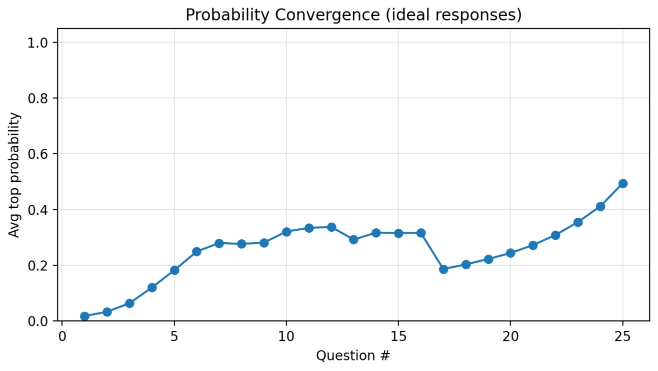

Fig. 7. Average convergence of the top candidate probability for ideal responses. Convergence is smooth and rapid.

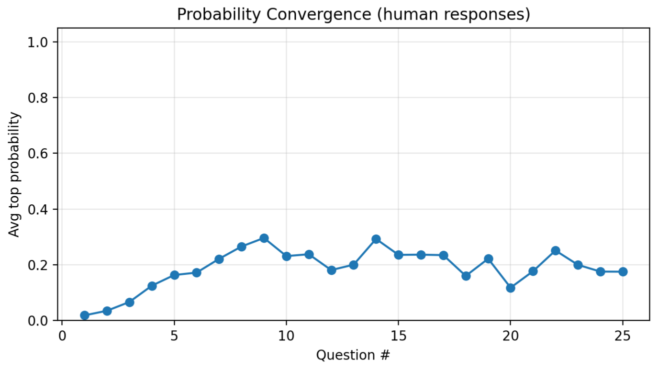

Fig. 8. Average convergence of the top candidate probability for human-like responses. Noise and slower convergence are clearly visible.

VI. Conclusion and Future Work

Ind-inator demonstrates that a combination of probabilistic inference, graded answers, and entropy-based, decision-tree-style question selection can support an effective and interpretable character-guessing game. The script-based data pipeline for characters, traits, and questions simplifies maintenance and extension of the dataset, and the modular design allows the same engine to drive both web and desktop interfaces.

Future work may include:

- Expanding the character set and trait taxonomy to reduce ambiguity among similar characters.
- Adding more fine-grained traits and refining questions for known difficult cases.
- Implementing more systematic evaluation, including large-scale automated simulations with quantitative performance statistics.
- Exploring alternative or complementary question-selection strategies, such as clustering-based approaches or precomputed decision trees for particular subsets of characters.

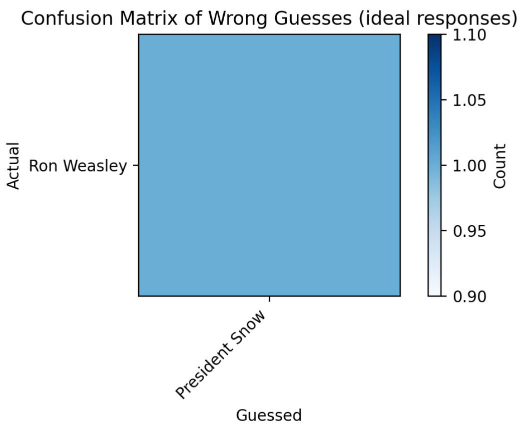

Fig. 9. Confusion matrix of incorrect guesses under the ideal model. Only one misclassification occurred: Ron Weasley as President Snow.

VII. Contribution

Although all team members collaborated on the overall design, implementation, debugging, and testing of Ind-inator, certain areas had primary leads.

Jay Jagdishkumar Bava took a leading role on the back-end, including game state management, integration of the AI engine with the Flask API, handling probability-update flows, and contributing to debugging and performance tuning.

Inderpreet Singh Doad also led backend work, including the design of the probabilistic inference approach, the entropy-based and phase-based question-selection logic, the data and question-generation scripts, and the integration of the AI engine into both the backend and user interfaces, as well as preparation of the written report.

Emily Jordan Berlinghoff focused primarily on the React front-end, API integration, and UI/UX design, while also contributing to backend discussions, testing, and debugging to ensure consistent behaviour across the stack.

Vidhi Joshi focused on character and trait definition, validation of generated questions, and manual testing, while also participating in design decisions and providing feedback on both backend behaviour and front-end interaction flows.

All team members participated in design discussions, implementation decisions, and testing, and adhered to the course expectations for professionalism and academic integrity.

References

[1] Akinator, "Akinator the Web Genius," 2024. [Online]. Available: https://en.akinator.com/
[2] DeepLearning.AI, "AI Education and Learning Resources," 2024. [Online]. Available: https://www.deeplearning.ai/
[3] A. Ronacher, "Flask (Python Web Framework)," 2010. [Online]. Available: https://flask.palletsprojects.com/
[4] Google, "TensorFlow: An end-to-end open source machine learning platform," 2024. [Online]. Available: https://www.tensorflow.org/
[5] Hugging Face, "Hugging Face: Open-source AI models and tools," 2024. [Online]. Available: https://huggingface.co/
[6] D. Mackay, *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press, 2003. Available online: https://www.inference.org.uk/mackay/itila/

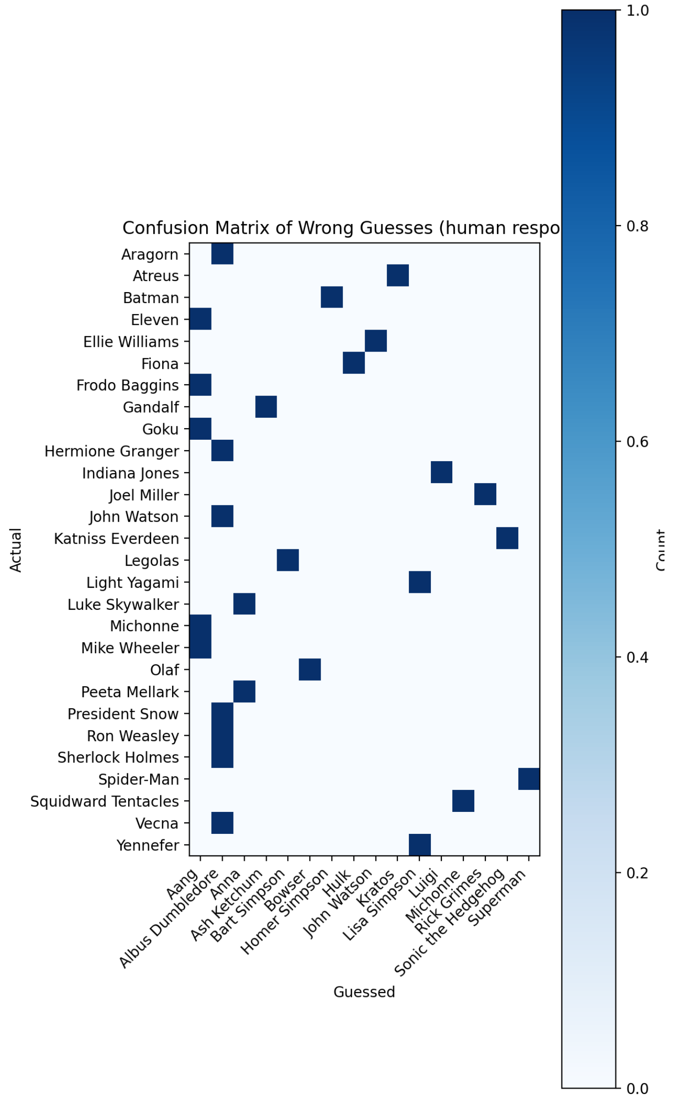

Fig. 10. Confusion matrix of incorrect guesses under the human-like model. Darker cells indicate repeated confusions between specific character pairs.

[7] C. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge University Press, 2008. Available online: https://nlp.stanford.edu/IR-book/

[8] MIT, “Introduction to Deep Learning,” 2024. [Online]. Available: https://introtodeeplearning.com/

[9] Meta AI, “PyTorch: An open source machine learning framework,” 2024. [Online]. Available: https://pytorch.org/

[10] Meta Platforms, “React: A JavaScript Library for Building User Interfaces,” 2024. [Online]. Available: https://react.dev/

[11] scikit-learn Developers, “scikit-learn: Machine Learning in Python,” 2024. [Online]. Available: https://scikit-learn.org/