Model & Utility Application

Farida Bey
900212071

&

Youssef Ghaleb
900211976

INTRODUCTION

This phase of the project focuses on implementing the final model and client-server application. The primary goals include creating a seamless user interface for data input and feedback, efficient preprocessing for model prediction, and robust APIs to facilitate interaction between the client and server.

2. IMPLEMENTATION CHOICES

2.1 Model

The model training process employed focal loss to address class imbalance, which was identified as a significant issue in earlier experiments, leading to poor performance on underrepresented genres. Focal loss was chosen to emphasize harder-to-classify samples by reducing the influence of well-classified instances, with tunable parameters such as alpha (class weighting factor) and gamma (focusing factor) optimized using Keras Tuner. The primary objective metric for tuning was recall, selected to prioritize correctly identifying all relevant genres, especially for minority classes, as precision alone would not sufficiently address the imbalance problem.

The neural network architecture was designed with flexibility in mind, allowing for one or two hidden layers to prevent overfitting while maintaining sufficient model capacity. Each layer's size was also tunable, ranging from 32 to 256 units, with ReLU activation ensuring non-linearity. The output layer utilized a sigmoid activation function, which was chosen because it is well-suited for multilabel classification tasks, where each genre is predicted independently, with values between 0 and 1 representing the likelihood of each label. This setup allows the model to predict multiple genres for each movie, addressing the multilabel nature of the problem. Hyperparameter optimization was performed using Random Search and Hyperband, enabling efficient exploration of diverse configurations. This careful balance between model complexity and regularization ensured robust performance across all genres, particularly for those with fewer training samples.

2.2 Application Architecture

The application is designed with a client-server architecture, balancing computational workloads efficiently. The **client**handles user interactions and performs lightweight preprocessing tasks, such as gathering and formatting input data. It communicates with the **server** through HTTP requests using a Flask-based API. The **server** processes these requests, leveraging its computational resources for tasks like natural language preprocessing (e.g., tokenization, stemming, stopword removal) and executing machine learning models. This architecture optimizes performance by minimizing client-side complexity while ensuring the server handles intensive operations like database queries and neural network inference. The design also supports scalability, allowing the server to manage multiple client requests concurrently.

2.2.1 Graphical User Interface (GUI)

The application's GUI is built using Tkinter, emphasizing usability and accessibility. The interface includes an intro page with graphical elements such as images and employs widgets like dropdown menus and checkboxes to reduce manual input and prevent user errors. These controls guide users in selecting options efficiently, enhancing the experience for non-technical users. The layout is designed for clarity, ensuring essential features are easily accessible, while error handling mechanisms provide feedback through intuitive message boxes. This thoughtful design minimizes the risk of input errors and streamlines the interaction flow.

3. APIs

The server implements a RESTful API using Flask to handle interactions with the client application. Key endpoints include:

3.1 /classify (POST)

The /classify endpoint is the core of the application, responsible for handling classification requests. When a client sends a POST request to this endpoint, it typically includes input data such as a movie title, synopsis, or other descriptive information. Upon receiving the request, the server processes the data by applying preprocessing techniques, such as tokenization, stemming, and stopword removal, to prepare it for model inference.
Next, a trained machine learning model is invoked to classify the input into predefined categories, such as movie genres. The server then returns the classification result, often formatted as a JSON response, containing the predicted genre(s) along with associated probabilities or confidence scores.
Importance:
This endpoint is central to the application's functionality, as it bridges the gap between user input and the intelligent system's output. It ensures a seamless user experience by automating the complex process of data transformation and analysis, delivering actionable results in real-time.

3.2 /history (GET)

The /history endpoint provides access to historical data, allowing users or administrators to retrieve records of past classifications. When a GET request is made to this endpoint, the server queries its SQLite database to fetch relevant information, such as input data, predicted results, timestamps, or user-specific logs.
Importance:
This endpoint adds value by enabling tracking and analysis of previous predictions. For users, it offers a convenient way to review past interactions, while for developers or administrators, it aids in debugging, performance monitoring, and improving the machine learning model based on historical data trends.

- The server authenticates and validates the GET request.
- A database query is executed to retrieve the required records.
- The data is formatted into a structured response, such as a JSON object, which is then returned to the client.

4. Preprocessing

The preprocessing pipeline is designed to mirror the transformations applied during the training phase, ensuring consistency and accuracy in predictions.
Text data undergoes extensive preprocessing, including tokenization, stemming, stopword removal, and embedding. The embeddings are generated using a pre-trained Word2Vec model with a 300-dimensional vector space, capturing semantic relationships between words and phrases to provide meaningful numerical representations for the machine learning model.
Numerical features are scaled based on their distributions, which are recalculated dynamically from the incoming data. This ensures that the preprocessing adapts to potential variations in input distributions, maintaining compatibility with the model's requirements while preserving flexibility for unseen data patterns.
Categorical encoding is handled on the client side, where categorical variables are transformed into numerical formats using techniques like one-hot encoding or label encoding. This design choice reduces transmission latency. By delegating this simpler preprocessing task to the client, the server can focus on computationally intensive operations like embeddings, scaling, and inference. This division of labor not only optimizes performance but also ensures a responsive user experience while adhering to established training standards.

5. Retraining Process and Current Limitations

The application includes functionality for retraining the machine learning model to improve performance by incorporating new data. This process involves fetching training data from an SQLite database, preprocessing it to align with the original training pipeline, and updating the model using TensorFlow and Keras. The retraining script is designed to load the existing model, fine-tune it with additional data, and save the updated version for continued use. However, the retraining process is not functional in its

current state because the database required to store and retrieve training data has not been initialized. Without a properly set up and populated database, the retraining pipeline cannot fetch the necessary inputs or proceed with the model update. While the code framework for handling retraining exists, it remains incomplete and non-operational due to this critical dependency.