Assignment 1: Design Document

By: Pradyumna Seethamraju

---

1. Model Staleness Monitoring

The code includes `apscheduler`, specifically the `BackgroundScheduler`, which can be configured to monitor model staleness by scheduling regular evaluations of the model's performance on validation data. Although not explicitly configured for staleness in the current code, the `BackgroundScheduler` could be set up to:

- **Run Evaluations Periodically:** The `evaluate_model` function calculates accuracy and F1 score on the validation set. Running this function on a scheduled basis can help detect when the model's performance starts to degrade.
- **Threshold Alerts**: The threshold (defined as `THRESHOLD = 0.8`) can indicate a staleness trigger. If the performance metric, such as F1 or accuracy, falls below this threshold, an alert or flag could be triggered to indicate that the model may require retraining.

Key Configurations

- **Threshold Setting** (`THRESHOLD = 0.8`): Acts as a performance baseline. If metrics fall below this, the model is flagged as potentially stale.
- **Scheduler Frequency**: The `BackgroundScheduler` interval could be daily or weekly, depending on data volume and model retraining frequency needs.

2. Deployment Infrastructure

The current infrastructure setup is local, using Flask as a REST API server to serve predictions.

- **Local Hosting**: The Flask API is hosted locally using `Flask` to manage REST requests. It is configured to listen on `host='0.0.0.0'` and `port=5000`, allowing access on the specified port for local or LAN use.
- **Caching**: `Flask-Caching` is used to store responses temporarily, reducing redundant computations and lowering latency for repeated requests.
- **API Rate Limiting**: The `Flask-Limiter` library applies rate limits on requests per IP address, helping prevent excessive usage and optimizing local resource utilization.

Deployment Steps:

1. **Flask Server Setup**: The Flask app is initialized and runs locally on `0.0.0.0:5000`.
2. **Logging and Monitoring**: Basic logging is in place, with MLflow configured for experiment tracking and version control to track model performance and manage updates.
3. **Scheduler Execution**: The staleness monitoring runs periodically in the background via `apscheduler`, enabling periodic performance checks on a local schedule.

3. Inference Methods and Cost Optimizations

The system provides two inference methods: live (single) and batch prediction.

3.1. Live Inference (Single Prediction)

- **Route**: `/predict`
- **Process**: Accepts a single text in JSON format, tokenizes it, and runs it through the model.
- **Optimization**: Although each request is processed individually, caching responses for similar inputs can reduce redundant computation, especially for repeated inputs in real-time usage.

3.2. Batch Inference

- **Route**: `/batch_predict`
- **Process**: Accepts a list of texts in JSON format, tokenizes them, and processes them in batches.
- **Optimization**: Batch processing maximizes local computational efficiency, especially for GPU-based inferences. Using an appropriate batch size (e.g., `batch_size=32` as per code) ensures efficient memory utilization and lowers latency per request.

Additional Cost Optimization Techniques:

- **Caching of Results**: The `Flask-Caching` integration helps store predictions for repeated requests, reducing unnecessary recalculations.
- **Batch Inference**: The batch mode improves efficiency by grouping requests, reducing per-request overhead and optimizing computational resource usage.
