P300 Brain Invaders

1. Overview

This document describes the design of an MLOps system built for a P300-based Brain-Computer Interface (BCI) application. The core function is to classify single-trial electroencephalography (EEG) signals to detect the P300 event-related potential (ERP), which indicates a user attending to a specific visual stimulus. Utilizing the publicly available bi2014a dataset (https://hal.science/hal-02171575), recorded from 71 subjects playing a visual P300 BCI game ("Brain Invaders") with 16 EEG channels, the project will develop a modular, microservice-oriented pipeline covering data ingestion, preprocessing, model training, and real-time (simulated) prediction serving. The primary outcome is a demonstration of key MLOps principles and infrastructure implementation on a neurophysiological signal processing and classification task.

2. Motivation

Brain-Computer Interfaces provide a crucial communication and control channel for individuals with severe motor impairments. Among various BCI paradigms, the P300 Speller is widely recognized for its potential to enable reliable communication through visual attention. The P300 is a positive-going ERP component elicited approximately 250-600ms after a rare, task-relevant stimulus, such as a flashing character or group of characters a user is focusing on. By accurately detecting the presence of a P300 response in the EEG following a flash, a BCI system can infer the user's intended selection.

Moving BCI technology from laboratory demonstrations to practical, deployable systems requires addressing significant engineering challenges beyond just model development. This is where MLOps might become essential. Building automated, reliable, and scalable pipelines for data handling, model training, deployment, and monitoring is useful for the robustness and maintainability of BCI applications.

This project will serve me as a practical case study to apply MLOps principles – such as automated pipelines, service-oriented architecture, and monitoring – to a real neurophysiological dataset (bi2014a). By focusing on the infrastructure around the ML model, I hope to gain the experience in deploying and managing ML systems in a domain with specific requirements like low-latency prediction. I will practice with Docker, which I have never used before (actually, started using it recently on the Data Streaming with Kafka course).

.

3. Success metrics

ML model performance check

- Primary metric for evaluating binary classification performance is AUC (Area under the ROC curve), robust to class imbalance (Non-Targets are more frequent than Targets). Aim for AUC significantly above chance (0.5).
- Measures agreement between predictions and ground truth, accounting for random chance. Aim for Kappa > 0.4 (moderate agreement) or higher (Still think about its usage).
- Overall percentage of correct Target/Non-Target classifications.
- Latency for a single epoch prediction below a defined threshold (e.g., < 200ms) to simulate responsiveness required by a BCI.

MLOps setup:

- Clear system decomposition into microservices with defined roles and communication.
- Successful implementation of data pipelines (ingestion, processing, training data prep).
- Containerization of core services using Docker.
- Basic implementation of a monitoring setup for service health and key metrics.

4. Requirements & Constraints

Functional Requirements:

- The system will ingest EEG data from the bi2014a dataset files (.mat/.csv).
- The system will preprocess raw EEG data (filtering, epoching, feature extraction) into a format suitable for the ML model.
- The system will train a binary classification model on processed EEG epochs from the dataset.
- The system will host the trained model via an API for prediction requests.
- The system will accept processed EEG epoch features as input to the prediction API and return a Target/Non-Target classification or probability.
- The system will provide a mechanism to save and load trained model weights.

Non-functional Requirements:

- The system must process prediction requests with low latency (target p95 < 200ms). The preprocessing and serving services should handle simulated throughput equivalent to the flashing rate of the P300 speller paradigm.
- Services should include basic error handling and logging.

- The system should be modular, with services containerized and dependencies managed.
- Basic monitoring and logging should be implemented for key services.

Constraints:

- Must use the bi2014a dataset.
- Real-time EEG data acquisition is out of scope; data will be processed from pre-recorded files, simulating a real-time stream.
- The full P300 Speller application logic (e.g., accumulating flash predictions to select a character, graphical interface) is out of scope. The focus is on the ML pipeline and prediction service.
- Complex BCI-specific challenges like subject-to-subject variability, online adaptation, and sophisticated artifact handling may be simplified or scoped out.

4.1 What's in-scope & out-of-scope?

In-scope

- Reading and loading data from bi2014a files (.mat/.csv).
- Implementing a defined EEG preprocessing pipeline (filtering, epoching, baseline correction, feature extraction) suitable for P300 classification.
- Developing and training a binary classifier for Target vs. Non-Target epochs.
- Building and deploying microservices for data ingestion, preprocessing, model training, model serving, and monitoring.
- Implementing communication patterns between services (e.g., API, message queue/shared storage simulation).
- Containerizing services using Docker.
- Establishing a basic monitoring setup (metrics and logging).
- Simulating the flow of data through the system to test the pipeline and serving latency/throughput.
- Storing and versioning model artifacts (basic model registry).

Out-of-scope:

- Real-time EEG data acquisition hardware integration.
- Development of a full interactive P300 Speller graphical user interface or application.
- Clinical validation or testing with real users with disabilities.
- Handling all subjects from the bi2014a dataset, especially those noted as having poor data quality, might be simplified (e.g., focusing on a subset of good subjects initially).
- Sophisticated online learning, model adaptation, or calibration procedures.

- Advanced data validation (beyond basic format/completeness checks).
- Production-grade security.

5. Methodology

5.1. Problem statement

The core machine learning problem is framed as a supervised binary classification task. Given a segment (epoch) of multi-channel EEG data recorded from a participant immediately following the visual flash of a symbol group in the "Brain Invaders" P300 speller interface, classify whether this flash was a 'Target' flash (i.e., the user was reacting to a symbol within that flashed group) or a 'Non-Target' flash (i.e., the user was not reacting to any symbol in that group). The output of the model will typically be a probability or a binary label (Target/Non-Target) for each epoch.

5.2. Data

The project will exclusively use the bi2014a dataset. Publicly available at https://doi.org/10.5281/zenodo.3266223. Contains data from 71 subjects (though data quality varies, and some subjects are noted as noisy or having inconsistent ERPs – strategy for handling this is in 6.9 Risks & Uncertainties / 4.1 Scope). There are 16 active dry EEG electrodes (Fp1, Fp2, F5, AFZ, F6, T7, Cz, T8, P7, P3, PZ, P4, P8, O1, Oz, O2), plus reference and ground. The 16 signal channels will be used for classification. Sampling rate is 512 Hz.

Paradigm is visual P300 "oddball" paradigm (sequences of repetitive stimuli are infrequently interrupted by a deviant stimulus) within the "Brain Invaders" game. Symbols flash in groups. Each repetition consists of 12 flashes, with 2 being 'Target' flashes (containing the symbol the user is focused on) and 10 being 'Non-Target' flashes.

The data is provided in `.mat` and `.csv` formats. Time-series EEG data along with event markers indicating the timing and type (Target/Non-Target) of each flash.
The system will expect processed feature vectors derived from single EEG epochs corresponding to individual flashes. The format will be a fixed-size numerical array (a NumPy array).

5.3. Techniques

Preprocessing

1. Load raw EEG data and event markers from `.mat` or `.csv` files (using libraries like `scipy.io.loadmat` or pandas, potentially integrated with MNE-Python capabilities if converting formats).

2. Select the 16 EEG signal channels, excluding reference/ground.
3. Apply a band-pass filter to the continuous EEG data to isolate the frequency range relevant for P300 (1 Hz to 20 Hz - since it's a visual task). A notch filter might also be applied at the power line frequency (50 or 60 Hz – don't know yet).
4. Segment the continuous filtered data into short time windows (epochs) time-locked to the onset of each visual flash event. A common epoch window for P300 is from a short pre-stimulus baseline to ~600-800ms post-stimulus (e.g., -100 ms to 600 ms relative to flash onset). Event markers from the data will be used to identify flash timings and their Target/Non-Target labels.
5. Subtract the mean amplitude of the pre-stimulus baseline period (-100 ms to 0 ms) from the entire epoch for each channel.
6. Implement simple automatic artifact rejection by excluding epochs where the amplitude on any channel exceeds a high threshold (+/- 100 microV), as these often indicate eye blinks or muscle movements that could obscure the P300.
7. As for feature extraction, a common approach for P300 with a moderate number of channels is to downsample the epoched data. The cleaned epochs (time x channels) are downsampled to a lower frequency (basically 100 Hz or 64 Hz). The values across all channels at the downsampled time points are then concatenated into a single feature vector. For an epoch of length T ms with C channels, downsampled to F Hz, the feature vector size will be approximately `C * (T/1000 * F)`. This captures the time-course information across electrodes. I am still deciding what to do next, and if I need to fragmentize the data more.

Model

Regularized Linear Discriminant Analysis (LDA) classifier or Ridge Regression are fine and commonly used models for P300 classification, especially with highly homogeneous data like this. They are computationally efficient for both training and inference, which aligns well with MLOps goals for serving latency. I will prioritize Ridge Regression or Shrinkage LDA for initial implementation. The chosen model will be trained on the extracted feature vectors and their corresponding binary labels (Target=1, Non-Target=0).

5.4. Experimentation & Validation

Validation

To simulate generalization and account for subject variability, I will use subject-wise cross-validation approach. Data will be split such that the model is trained on data from a set of subjects and evaluated on data from hold-out, unseen subjects. Alternatively, if focusing on a single subject, stratified k-fold cross-validation will be used to ensure the ratio of Target/Non-Target epochs is maintained in each fold.

Evaluation

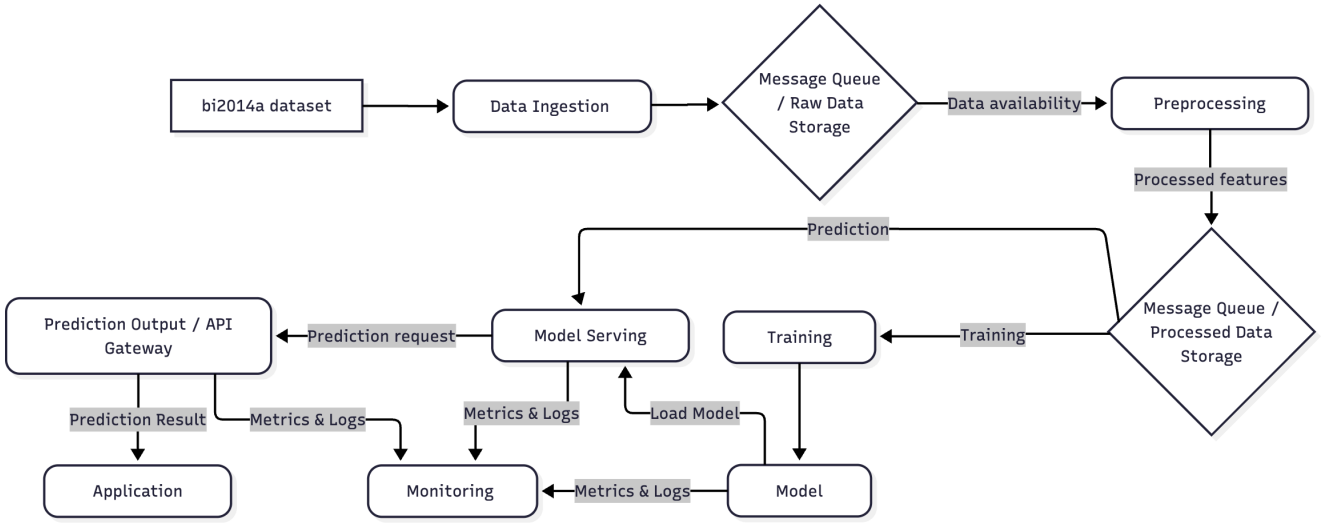
- AUC will be calculatedfrom predicted probabilities, providing a threshold-independent measure of the model's ability to discriminate between Target and Non-Target epochs.
- Kappa score will be calculated from predicted binary labels (using a threshold, typically 0.5 probability), assessing agreement beyond chance.
- Overall proportion of correct classifications.
- Confusion matrix to understand performance breakdown.
- Different preprocessing parameters (filter ranges, epoch windows, downsampling rates) and model hyperparameters will be experimented with offline using the validation strategy to select the best performing configuration before deploying the model.

5.5. Human-in-the-loop

While the core MLOps system focuses on the automated pipeline and model serving, a real P300 BCI involves a human user whose brain signals are being interpreted. In the context of this project, the "human-in-the-loop" aspect is primarily represented by the source data. The system's performance is directly tied to the human's ability to generate discernible P300 responses and the model's ability to decode them. For this MLOps project's scope, there is no active human feedback loop within the system design itself (I don't plan on implementing online model adaptation). The project only simulates serving predictions to a downstream application that would typically interact with a human user.
Okay, let's fill out Section 6 (Implementation) of your design document, describing how the P300 BCI MLOps system will be built.

6. Implementation

6.1. High-level design



- Raw data files are processed by the `Data Ingestion Service`, pushed to a queue/storage.
- The `Preprocessing Service` consumes this, applies transformations, and stores processed features/labels.
- The `Training Service` uses processed data to train models, storing results in the `Model Registry`.
- The `Model Serving Service` loads models from the registry and provides predictions via an API, consuming processed data (simulated as new input). The `Application Simulator` consumes these predictions.
- `Monitoring Service` collects metrics and logs from other components. Services interact with the `Model Registry` to save/load models.

Microservice breakdown

1. **`data-ingestion-service`** will be responsible for reading the raw EEG data files from the bi2014a dataset. Reads `.mat` or `.csv` files, extracts the raw EEG time series and event markers (flash timings and types). Its primary output is making the raw data available to the Preprocessing Service.
2. **`preprocessing-service`** will perform all necessary signal processing and feature engineering on the raw EEG data. Listens for or pulls raw data segments provided by the Ingestion Service. Does the preprocessing and then stores the resulting feature vectors and their corresponding Target/Non-Target labels in a structured format (Parquet) in processed data storage.
3. **`training-service`** manages the machine learning model training process. Triggered periodically or manually (via an API call). Reads the processed feature vectors and labels from the processed data storage. Splits data for training and validation. Trains the chosen binary classifier model (LDA, Ridge). Evaluates the model's performance using offline

metrics (AUC, Kappa, Accuracy). Serializes the trained model object and saves it, along with evaluation metrics and parameters, to the Model Registry.

4. **`model-serving-service`** hosts the trained ML model and provides a real-time prediction endpoint. Upon startup or model update signal, loads the latest or a specified version of the model from the Model Registry. Exposes a REST API endpoint that accepts a feature vector for a single EEG epoch as input. Runs the loaded model inference on the input feature vector. Returns the predicted class (Target/Non-Target) or prediction probability. This service is critical for low-latency predictions.
5. **`model-registry-service`** is a central repository for storing, versioning, and managing trained machine learning models and their metadata. Provides APIs for the Training Service to register/save new model versions (including performance metrics, training data reference, parameters) and for the Model Serving Service to discover and load specific model versions. Stores the actual model files (using pickle) in persistent storage.
6. **`monitoring-service`** collects, aggregates, and visualizes operational metrics and logs from all other services. Scrapes metrics endpoints exposed by other services or receives metrics pushed to a gateway. Stores metrics in a time-series database, showing service health (CPU, memory, uptime), request rates, error rates, data processing counts, and prediction metrics. Collects and centralizes logs.
7. **`application-simulator-service`**: simulates a downstream BCI application that would consume predictions from the Model Serving Service. Generates or reads sample/simulated processed feature vectors (representing new flashes in a BCI session). Makes API calls to the `model-serving-service` endpoint with these features. Receives and prints/logs the prediction results, simulating how a real application would use the output.

6.2. Infra

The system will be implemented using standard tools and technologies, primarily focusing on Python for development and Docker for containerization and orchestration.

Libraries

- Data: `mne-python`, `numpy`, `pandas`, potentially `scipy.io` for `.mat` files.
- ML model: `scikit-learn` (for LDA, Ridge) or `pyriemann` (for covariance features, MDM) based on the chosen methodology or fallback plan.
- Web framework for APIs: `FastAPI` or `Flask` for building the REST interfaces (`model-serving-service`, `model-registry-service` if separate API).
- Messaging queue: Redpanda. A simple approach using shared storage might be sufficient for a course project.
- Storage: a shared volume in a container.
- Monitoring: Weights&Biases.

- Orchestration: `Docker` will be used to containerize each microservice. `Docker Compose` will be used for defining and running the multi-container application locally.

6.3. Performance (Throughput, Latency)

Latency

The Model Serving Service is the most latency-sensitive component. The choice of linear models (LDA/Ridge) or simple Riemannian methods (MDM) operating on relatively compact feature vectors is intentional to ensure low inference time per epoch. Preprocessing steps will also be optimized for speed within the Preprocessing Service. The API framework (`FastAPI`) is chosen for its asynchronous capabilities and high performance.

Throughput

The Preprocessing and Model Serving services will be developed to be stateless (or manage state externally), allowing multiple instances to run in parallel to handle a higher rate of incoming data/requests. Docker Compose can simulate this using the `--scale` option.
While individual prediction latency is key, the speed of the full pipeline (Ingestion -> Preprocessing -> Serving) determines how quickly data could be processed and predicted. The asynchronous nature of the pipeline improves overall throughput.

6.4. Security

Given the scope of a course project, security will be addressed at a basic level appropriate for demonstrating concepts, not production readiness.

- Services will run within a local Docker network, not exposed directly to the public internet.
- Inter-service communication within the Docker network will be assumed trustworthy for this project.
- Access to the Model Serving API could potentially simulate basic authentication (idk, like a simple shared API key), but this is not my primary focus.
- Data storage will be within the project's controlled environment (local disk mounted into containers).

6.5. Data privacy

This project uses a publicly available, anonymized dataset (bi2014a).

6.6. Monitoring & Alarms

Monitoring

- Service operational metrics like CPU/Memory usage (from Docker/host), request count, error rates, latency (for API endpoints), service uptime.
- Number of raw files/batches processed by Ingestion, number of epochs processed/rejected by Preprocessing, number of prediction requests handled by Serving.
- Model performance metrics like accuracy/AUC over time would require ground truth feedback in real-time, which is out of scope, but metrics related to the rate of predictions are feasible.
- Services will expose metrics in a format Weight&Biases or Prometheus can scrape. Basic logging within services will be implemented, viewable via `docker logs`.

Alarming

A service container unhealthy/restarting frequently, zero prediction requests for a prolonged period) can be conceptually defined and potentially set up.

6.7. Cost

The primary cost for this project is the computational resources on the student's local machine running Docker. There are no anticipated cloud infrastructure costs within the defined scope using Docker Compose locally.

6.8. Integration points

- Upstream: file system access to the downloaded bi2014a dataset files (.mat/.csv).
- `Data Ingestion` -> `Preprocessing`: message queue or by writing raw data chunks to shared storage and notifying Preprocessing.
- `Preprocessing` -> `Training` / `Serving`: writing processed features/labels to shared storage (Parquet files) and potentially sending messages indicating new data availability.
- `Training` / `Serving` <-> `Model Registry`: REST API calls to save/load model artifacts.
- Services -> `Monitoring`: pushing metrics.
- Downstream: `Model Serving Service` exposes a REST API endpoint which will be consumed by the `Application Simulator`.

6.9. Risks & Uncertainties

- The bi2014a dataset abstract notes variability and quality issues in some subjects. This could impact model performance if not adequately addressed through preprocessing or subject selection.
- Ensuring consistent data formats (especially for processed features) passed between services is crucial and can be challenging. Serializing and deserializing complex data structures like covariance matrices (if the fallback is used) adds complexity.

- Getting all services to correctly communicate and the overall pipeline to run smoothly end-to-end within Docker Compose can involve debugging network configurations, volume mounts, and service startup order.

7. Appendix

7.1. Alternatives

This section discusses alternative approaches considered for the data preprocessing and machine learning components compared to the primary methodology described in Section 5.3.

The **primary methodology** (chosen) involves:

- Preprocessing: band-pass filtering (1-20Hz), epoching (-100 to 600-800ms relative to stimulus onset), baseline correction, basic amplitude-based artifact rejection.
- Feature extraction: downsampling the epoched time-series data and concatenating samples across channels to form a single feature vector per epoch.
- ML model: training a linear classifier such as Regularized Linear Discriminant Analysis (LDA) or Ridge Regression on the feature vectors.
  This approach was chosen primarily because:
- It provides a standard vector-based feature representation which simplifies data handling and integration between services.
- LDA and Ridge Regression are computationally efficient for both training and inference, and I had experience with them in the previous BCI project.

An **alternative methodology** (considered), similar to the approach potentially used or suggested by the authors of the bi2014a dataset based on provided code examples, involves:

- Preprocessing: similar filtering (1-24Hz), epoching (0-800ms).
- Feature extraction: estimating covariance matrices for each epoch, using techniques like xDAWN (Eigenvalue Decomposition of Average ERPs with Noise). This belongs to the family of Riemannian geometry-based methods.
- ML model: using a classifier designed for Riemannian features, such as Minimum Distance to Mean (MDM) or tangent space mapping followed by a standard classifier.

Pros of the alternative:

- Often considered state-of-the-art or highly competitive in BCI classification tasks, particularly for single-trial classification.
- Effectively leverages the spatial covariance structure of multi-channel EEG data.
- Can be more robust to certain types of noise distributions.
  Cons of the alternative:

- Features are matrices (or points on a manifold) rather than simple vectors, which can complicate serialization, storage, and passing data between microservices using standard formats (might require specific handling or encoding, don't know yet as well).
- The underlying mathematical concepts (Riemannian geometry) are less commonly understood than standard linear algebra used in vector-based methods, so harder for interpretation.

Fallback plan:
If, during the implementation and offline validation phase, the chosen primary methodology (downsampling + LDA/Ridge) performs poorly (resulting AUC is close to chance level) or proves unexpectedly difficult to implement correctly and reliably within the project timeline, then I will revert to implementing the alternative methodology based on Riemannian features (covariance matrices) and a Riemannian classifier (MDM).