Group Members: Anita Gee, Siwei Guo, Yingzhu Chen, Nemo (Sorachat) Chavalvechakul

Group Name: Mental Health Warriors

Week 12 Report

Saving the Model for Deployment

The final XGBoost model was saved using Python's pickle module to ensure it can be reused or deployed without retraining. A directory named models was created (if it did not already exist) to store the model and related artifacts. The model filename was automatically generated by converting the model name to lowercase and replacing spaces with underscores. The file was then saved as a .pkl file using binary write mode.

In addition to the model, several key datasets were saved to support reproducibility and evaluation. These include the resampled training features and labels (X_train_resampled1, y_train_resampled1), validation data (val_x1, val_y1), and test data (test_x_combined1, test_y1). The predictions on the test set (y_test_pred) were also saved. All files were stored in the models directory using descriptive filenames for clarity and organization.

To verify the integrity of the saved files, each dataset and the model were reloaded using pickle. The shape of each reloaded object was printed to confirm that the data structures remained intact. This workflow ensures that the trained model and its supporting data can be reliably used for deployment or further analysis.

Documenting the Environment Dependencies

Accurate documentation of the development environment is essential for reproducibility and smooth deployment. The operating system used was identified via os.name, which returned {os_name}. The Python version was {python_version}, confirmed through sys.version.

The environment was managed using Conda, and the complete list of installed packages was obtained using the pip freeze command. The following libraries were directly relevant to the model training, evaluation, visualization, and deployment tasks:

- Core Libraries:
  numpy==1.26.4, pandas==2.2.3, scipy==1.13.1

- Machine Learning:
  xgboost==2.1.3, scikit-learn==1.6.1, imbalanced-learn==0.13.0, shap==0.47.1

- Natural Language Processing (NLP):
  nltk==3.9.1, wordcloud==1.9.4, regex==2024.11.6

- Visualization:
  matplotlib==3.10.0, seaborn==0.13.2, plotly==5.24.1

- System and Utility Libraries:
  os, sys, pickle, gc, joblib==1.4.2, threadpoolctl==3.5.0

- Deep Learning and TensorFlow Support (installed but not used in this specific task):
  tensorflow==2.18.0, keras==3.8.0, h5py==3.12.1

- Other Libraries Installed in the Environment:
  Many additional packages were included as part of the working environment (e.g., Jupyter-related packages, Conda utilities, and support libraries such as protobuf, requests, pyyaml, jsonschema, etc.). While not directly used in this task, they contribute to the full configuration and execution context.

This detailed list of dependencies ensures the project environment can be fully recreated. If deployment is needed on a different system, all necessary packages and versions are clearly specified to avoid compatibility issues. For practical use, the pip freeze output may be stored in a requirements.txt file and used to recreate the environment via pip install -r requirements.txt.

Real‑Time Inference vs. Batch Scoring

For our XGBoost classification model—designed to deliver immediate, personalized insights in response to each new data submission—it makes the most sense to adopt a real‑time inference architecture rather than a purely batch scoring approach. In a real‑time setup, the trained model is exposed behind a lightweight RESTful API or gRPC endpoint. As soon as a user or upstream service emits a new feature vector (for example, a freshly completed risk assessment form, transaction record, or behavioral signal), the inference service computes and returns a prediction within milliseconds. This low‑latency round‑trip is critical if downstream components—be they front‑end dashboards, customer‑facing applications, or automated alerting systems—depend on “instant” feedback to drive user experience, trigger live workflows, or escalate urgent cases for human intervention.

By contrast, a batch deployment—where data is collected over fixed intervals (hourly, nightly) and scored en masse—would introduce unacceptable delays for time‑sensitive processes and degrade the end‑user experience. Although batch scoring can be more resource‑efficient at scale (processing millions of records in a big‑data pipeline), it is fundamentally episodic: no new data point is examined until the next scheduled run. In scenarios where rapid adaptation to evolving inputs is a business priority—such as fraud detection, clinical decision support, or personalized recommendations—batch latency often translates into missed opportunities or elevated risk. For these reasons, our production architecture favors real‑time inference: it ensures that every prediction reflects the very latest information, maximizes user engagement, and enables dynamic, feedback‑driven applications that would be impossible under a batch‑only regime.

Key Metrics and Their Importance

Once our model is live, mere uptime isn’t enough. We need a living dashboard of indicators that safeguard predictive quality, guard against data anomalies, confirm business value, and keep the service running smoothly. To that end, our monitoring framework weaves together four critical strands:

First, we continuously audit model behavior. By watching core performance statistics—such as accuracy on recently labeled examples, precision and recall against approval thresholds, and AUC‑ROC for overall separability—we get an early alarm if predictions start to drift. We also track probability calibration so that a “70% risk” score truly maps to a 70% observed hit rate. This ensures downstream decision‑makers can trust the raw scores, not just the binary labels.

Second, we guard against data drift. Because shifting user behavior or changes in our feature‑generation pipelines can stealthily undermine accuracy, we compute distributional distances between incoming feature streams and the training set. In practice, that means: measuring statistical divergences for each feature; watching for sudden jumps in the proportion of flagged predictions; and alerting if any upstream schema tweaks or missing columns appear. Such vigilance lets us intervene—retrain, roll back, or correct pipelines—before business processes start to misfire.

Third, we correlate predictions with business KPIs. It’s not enough that the model is technically sound; it must deliver commercial impact. For a fraud model, we compare the number of confirmed fraud cases prevented or chargebacks avoided; for a marketing model, we track lift in click‑through or conversion rates. Every week, we quantify “dollars saved” or “revenue generated” alongside model metrics. When the ROI curve flattens, it can be time to revisit feature engineering or explore new data sources.

Finally, we monitor system health. An ideal prediction can’t help if API latency spikes or GPU memory leaks crash the service. Our infrastructure dashboards include: p95 and p99 inference latencies to ensure responses stay under 200 ms, end-to-end error rates so we catch dependency failures, and resource utilization (CPU, GPU, RAM) to auto-scale before saturation.

By uniting these four streams—model signals, data-drift alarms, business-outcome metrics, and system health indicators—we build a 360° view of production health. Automated alerts tie each of these to clear runbooks: roll back a bad deploy, trigger a data-pipeline check, convene a cross-functional KPI review, or spin up more compute. This holistic approach not only preserves model accuracy but also ensures the ML investment consistently drives measurable business value.

Model Behavior Metrics:

Regarding monitoring a live model, we can define potential thresholds for green (nothing is wrong with the model), yellow (errors should be tracked closely), and red (model should be pulled out of production) for the selected metrics discussed above:

- **Accuracy, Precision, Recall, AUC-ROC**:
    - **Green**: The metric remains within a pre-defined acceptable range, indicating stable performance. For example, if the validation accuracy was 0.80, a **green threshold** could be maintaining an accuracy **above 0.78** on recently labeled data.
    - **Yellow**: The metric shows a statistically significant drop or falls below a warning threshold, indicating potential performance degradation that needs close monitoring. For instance, accuracy falling **between 0.75 and 0.78** could be a **yellow threshold**.
    - **Red**: The metric falls below a critical threshold, suggesting the model is no longer providing reliable predictions and should be pulled out of production. An accuracy **below 0.75** could be a **red threshold**.
- **Probability Calibration**:
    - **Green**: The predicted probabilities closely align with the observed hit rates. For example, if a "70% risk" score is consistently observed to correspond to approximately a 70% hit rate, this is **green**.
    - **Yellow**: There is a noticeable deviation between predicted probabilities and observed outcomes, potentially impacting the trust in the raw scores. For instance, a consistent

    deviation of **more than 5%** (e.g., a "70% risk" consistently resulting in only 60-65% hit rate) could be **yellow**.
    - **Red**: The model is significantly miscalibrated, making the probability scores unreliable for decision-making. A consistent deviation of **more than 10%** could be **red**.

System Health Metrics:

- **p95 and p99 Inference Latencies**: The source mentions a target of staying under 200 ms.
    - **Green**: Latencies are comfortably below the target. For example, **below 100 ms** could be **green**.
    - **Yellow**: Latencies are approaching or occasionally exceeding the target, indicating potential slowdowns. Latencies **between 150 ms and 200 ms** could be **yellow**, requiring investigation and potential scaling.
    - **Red**: Latencies consistently and significantly exceed the target, leading to a poor user experience or service disruptions. Latencies **above 200 ms** should be considered **red**, warranting immediate action to restore performance.
- **End-to-End Error Rates**:
    - **Green**: Error rates are minimal and stable. For example, **below 0.1%** could be **green**.
    - **Yellow**: Error rates show a noticeable increase, indicating potential issues with the model or its dependencies. An error rate **between 0.1% and 1%** could be **yellow**, triggering monitoring and debugging.
    - **Red**: Error rates reach a critical level, indicating service instability or failure to provide predictions. An error rate **above 1%** would likely be a **red threshold**, requiring immediate intervention and potentially pulling the model.
- **Resource Utilization (CPU, GPU, RAM)**: The source mentions auto-scaling before saturation.
    - **Green**: Resource utilization is within a healthy operating range.
    - **Yellow**: Utilization consistently reaches high levels (e.g., **above 70-80%**), indicating a potential need for scaling resources to prevent performance degradation.
    - **Red**: Resource saturation (e.g., **approaching or at 100%**) is causing service crashes or severe performance issues, necessitating immediate scaling or rollback.

It's crucial to remember that these thresholds are illustrative. In a real-world deployment, these values should be determined based on Baseline performance established during validation, acceptable levels of degradation from a business perspective, the cost and impact of false positives (pulling a good model) and false negatives (keeping a bad model live) and the specific context of the mental health application and the potential harm of inaccurate predictions or slow responses.

Risk Mitigation Strategies for Green, Yellow, and Red Flags

We propose the following risk mitigation strategies for the flag-based threshold system:

Green Flags (Nothing is wrong with the model)

- **Continuous Monitoring:** The primary mitigation strategy for green flags is to maintain the current state through continuous monitoring of all key metrics (accuracy, precision, recall, AUC-ROC, probability calibration, inference latencies, error rates, resource utilization).
- **Regular System Health Checks:** Ensure that the underlying infrastructure (API, GPU, etc.) remains healthy and within acceptable operating ranges.
- **No Immediate Action:** The focus is on maintaining stability and proactively looking for any signs of potential degradation.

Yellow Flags (Errors should be tracked closely)

When a metric enters the yellow zone, it signals a potential issue that needs closer attention and might require preventative action. The mitigation strategies include:

- **Increased Monitoring Frequency:** Increase the frequency of monitoring for the specific metric that triggered the yellow flag to observe if the trend worsens.
- **Automated Alerts and Runbooks:** Yellow flags should trigger automated alerts that notify the relevant teams to investigate the issue. These alerts should be tied to clear runbooks outlining the initial steps for diagnosis.
- **Investigate Potential Causes:** Analyze potential reasons for the deviation. For model behavior metrics (e.g., accuracy drop), this could involve looking for:
    - **Data Drift:** Compare the distribution of incoming feature streams with the training data to identify any significant shifts. Statistical divergences or sudden jumps in flagged predictions should be investigated.
    - **Upstream Changes:** Check for any recent schema tweaks or missing columns in the input data pipelines.
    - **Dependency Failures:** Investigate if any upstream dependencies are experiencing issues, contributing to increased error rates or latency.

- **Resource Scaling (Preemptive):** If resource utilization (CPU, GPU, RAM) enters the yellow zone (e.g., consistently above 70-80%), consider preemptive auto-scaling to prevent saturation and maintain performance.
- **KPI Review (If Business Impacting):** If business KPIs start to show a flattening ROI curve alongside yellow flags in model metrics, convene a cross-functional KPI review to assess the commercial impact and potentially revisit feature engineering or explore new data sources.

Red Flags (Model should be pulled out of production)

Red flags indicate critical issues that require immediate and decisive action, potentially including taking the model out of production. The mitigation strategies include:

- **Automated Alerts and Immediate Action:** Red flags should trigger high-priority alerts and, in some cases, automated actions like rolling back a bad deploy.
- **Rollback to a Stable Version:** If the red flag is related to a recent model deployment (e.g., a sudden and significant drop in accuracy or a spike in error rates), the primary mitigation strategy is to immediately roll back to the last known stable version of the model.
- **Halt Real-Time Inference (If Applicable):** If the model is deployed for real-time inference and red flags indicate unreliable predictions or service instability (e.g., consistently high latency, critical error rates), temporarily halt real-time inference to prevent negative impacts on users or downstream systems.
- **Trigger Data Pipeline Correction:** If data drift or upstream schema issues are identified as the root cause of a red flag, trigger an immediate data pipeline check and correction process.
- **Resource Scaling (Emergency) or Rollback:** If resource saturation (approaching 100%) is causing service crashes or severe performance issues (red flag for resource utilization), initiate emergency auto-scaling or, if that's not immediately effective, roll back to a state with lower resource demands.
- **Post-Mortem Analysis:** After resolving the red flag issue, conduct a thorough post-mortem analysis to understand what went wrong and identify areas for improvement in the monitoring framework, deployment process, or model development lifecycle.

Discuss if and how frequently you would retrain your model and why?

Model retraining would not be a fixed schedule but rather triggered by specific events and observed performance degradations. For example:

- **In Response to Red Flags:** A red flag signifies a critical issue where the model is no longer providing reliable predictions. In such cases, immediate action is required, which could involve rolling back to a stable version and then investigating the root cause. If the degradation is due to significant data drift or model decay, retraining the model with more recent data or improved features would be a necessary step before redeploying.
- **In Response to Yellow Flags:** A **yellow flag** indicates potential performance degradation that needs close monitoring. When metrics fall into the yellow zone, it signals a need to **investigate the potential causes**, such as data drift, upstream changes, or dependency failures. If the yellow flags persist or worsen, and the investigation points to the model's decreasing ability to generalize to new data, **retraining would be considered a proactive mitigation strategy**. This is to intervene before the issues escalate to red flag levels and business processes start to misfire.
- **Detection of Data Drift:** Source explicitly states the importance of guarding against data drift by "computing distributional distances between incoming feature streams and the training set". When significant data drift is detected, it indicates that the statistical properties of the input data have changed over time. In such scenarios, **retraining the model on more recent data that reflects the new data distribution is crucial to maintain accuracy and relevance**. Automated alerts should trigger a data-pipeline check, and if the drift is substantial, retraining would be a key step in the correction process.
- **Following Model Improvements:** If there are significant improvements made to the model architecture, feature engineering, or preprocessing steps, tweaks through data drift monitoring are corrected, the model would need to be **retrained from scratch or fine-tuned** using these enhancements.

Data Drift and Concept Drift in Our Use Case

The data for this mental health sentiment analysis use case can definitely be impacted by both **data drift** and **concept drift**.

Data Drift:

Data drift occurs when the statistical properties of the input data change over time compared to the data on which the model was trained. In this context, this could manifest in several ways:

- **Shifting user behavior:** The language users employ to express their mental health concerns might evolve over time. New slang, abbreviations, or ways of describing feelings could emerge that were not present in the original training data.

- **Changes in feature-generation pipelines:** If the methods used to extract features from the text data (like TF-IDF or n-grams) undergo changes or if new features are engineered, the distribution of these features could shift.
- **Platform evolution:** If the platform from which the text data is sourced changes its format, policies, or user demographics, this could lead to alterations in the kind of statements being collected. For instance, the character limits or the nature of discussions on a platform could influence the length and content of user inputs.
- **Upstream schema tweaks or missing columns:** As mentioned in the context of data drift monitoring, changes in the data sources or pipelines that feed into the model could introduce schema changes or missing features.

Data drift can lead to a decrease in the model's predictive accuracy. If the model is not exposed to the new patterns and language used by users, it may fail to correctly identify the sentiment and mental health status expressed in their statements. This could result in an increase in false positives or false negatives, undermining the model's ability to predict well.

Concept Drift:

Concept drift refers to a situation where the relationship between the input features and the target variable (the mental health status) changes over time. . In our use case, this could happen because:

- **Evolving understanding of mental health:** Societal understanding, diagnostic criteria, and the way people talk about mental health conditions can change. What might have been considered a symptom of anxiety a few years ago might be understood differently now.
- **Shifting societal norms:** Changes in social stigma surrounding mental health could influence how openly and in what manner individuals discuss their feelings.
- **Impact of external events:** Major global events (e.g., pandemics, economic crises) can impact mental health and the language used to describe these experiences, potentially altering the signals associated with different mental states.

Concept drift can also significantly degrade the model's performance because the underlying relationships it learned during training are no longer valid. For example, if the linguistic cues associated with "stress" change due to a shift in how people perceive and discuss it, the model might misclassify these instances

Mitigation Strategies:

- **Continuous Monitoring of Model Behavior:** Implement a **living dashboard of indicators** to safeguard predictive quality. This involves continuously tracking core performance statistics such as **accuracy on recently labeled examples, precision and recall, and AUC-ROC**. Significant drops in these metrics (triggering **yellow or red flags**) could be an early sign of data or concept drift affecting the model's performance.
- **Monitoring Probability Calibration:** Track **probability calibration** to ensure that the model's risk scores are reliable. If the observed hit rate deviates significantly from the predicted probability, it could indicate a change in the underlying data or concept.
- **Guarding Against Data Drift:** Implement specific mechanisms to **compute distributional distances between incoming feature streams and the training set**. This includes:
    - **Measuring statistical divergences for each feature.**
    - **Watching for sudden jumps in the proportion of flagged predictions.**
    - **Alerting if any upstream schema tweaks or missing columns appear**.
    - These **data drift alarms** can trigger automated actions like a **data-pipeline check** to identify and correct issues before they severely impact the model.
- **Regular Model Retraining:** As discussed previously, model retraining should be an event-driven process where we retrain in response to red and yellow flags, detecting and fixing data drift and retraining after these issues are fixed using Data-Centric AI techniques.
- **Error Analysis:** Regularly conduct error analysis on both training and validation data to understand the types of mistakes the model is making. Changes in the patterns of errors could indicate drift affecting specific categories or linguistic features. Analyzing feature importance (using tools like SHAP) over time can also reveal if the key drivers of the model's predictions are shifting, which might be a sign of concept drift.
- **Version Control and Rollback Mechanisms:** Maintain version control of models and have the capability to **roll back to a stable version**. if a newly deployed model shows signs of poor performance due to unaddressed drift.