**BATTERY FAILURE PREDICTION PROJECT REPORT**

**Prepared by: SHERIN SAMUEL**

**Date: May 08, 2025**

**Introduction**

The project predicts battery failure using the NASA battery dataset, encompassing data processing, feature engineering, preprocessing, exploratory data analysis (EDA), class balancing with SMOTE, model development, hyperparameter tuning, SHAP analysis for interpretability, and deployment through a Flask API, Streamlit dashboard, and Power BI dashboard. The goal is to deliver a robust, interpretable solution for battery management.

**Data Processing and Feature Engineering**

The project starts by processing NASA .mat files (B0005, B0055, B0056) to extract discharge cycle data. Features like average voltage, current, temperature, and capacity are computed, alongside derived metrics: State of Charge (SOC) = (voltage - 3.0) / (4.2 - 3.0), State of Health (SOH) = (capacity / 2.0) \* 100, internal resistance (voltage drop / discharge current), and a failure flag (capacity < 1.4 Ah). Ambient temperatures are set (24°C for B0005, 4°C for B0055/B0056). The data is saved as nasa\_battery\_data\_combined.csv.

**Data Preprocessing**

Preprocessing ensures data quality by filling missing numerical values with medians, removing outliers using the Interquartile Range (IQR) method, encoding battery\_id with LabelEncoder, and standardizing features with StandardScaler. The resulting dataset, nasa\_battery\_data\_preprocessed.csv, has 370 rows and 12 columns with no missing values, ready for modeling.

**Exploratory Data Analysis (EDA)**

EDA generates visualizations including capacity degradation plots, correlation heatmaps, and failure label distributions. Key insights show capacity/SOH strongly correlate negatively with failure (-0.882548), while current correlates positively (0.778305). Plots are saved in eda\_plots, revealing degradation trends and feature relationships.

**Class Balancing with SMOTE**

To address class imbalance in the dataset (more failure instances: 66.49% failure vs. 33.51% no-failure), SMOTE is applied during training of Random Forest and XGBoost models. SMOTE oversamples the minority class ("No Failure") by generating synthetic samples, ensuring balanced training data and improving model performance on minority class predictions, as seen in the improved recall scores (e.g., 0.9796 for Random Forest "Failure").

**Model Development**

Multiple models are developed:

* **Random Forest**: Achieves 98.65% accuracy with SMOTE, with top features time (28.51%), capacity (27.09%), and voltage (22.49%).
* **XGBoost**: Also uses SMOTE, reaching 99% accuracy post-tuning.
* **One-Class SVM**: Detects anomalies with 96% accuracy, trained on non-failure data.
* **LSTM**: Handles sequential data (sequence length=20), achieving 97% accuracy with class weights.
* **Ensemble**: Combines XGBoost, One-Class SVM, and LSTM (weights: 0.5 LSTM, 0.3 XGBoost, 0.2 SVM), achieving 100% accuracy, though evaluation indicates potential overfitting (no "No Failure" predictions).  
  Predictions are saved in predictions (e.g., ensemble\_predictions.csv), and models in models (e.g., rf\_model.joblib).

**Hyperparameter Tuning**

Grid Search enhances model performance:

* **XGBoost**: Tunes max\_depth (3, 5, 7), learning\_rate (0.01, 0.1, 0.3), n\_estimators (100, 200), selecting max\_depth=3, learning\_rate=0.01, n\_estimators=100 (F1-score 0.99 for "Failure").
* **One-Class SVM**: Tunes nu (0.05, 0.1, 0.2), gamma (scale, auto, 0.1), selecting nu=0.05, gamma=scale (F1-score 0.97).
* **LSTM**: Manually tunes sequence\_length (10, 20) and units (50, 100), selecting sequence\_length=10, units=100 (F1-score 1.0).  
  Tuned models are saved (e.g., lstm\_model\_tuned.h5).

**SHAP Analysis for Interpretability**

SHAP (SHapley Additive exPlanations) analysis is applied to the Random Forest and XGBoost models to interpret predictions. For Random Forest, SHAP values confirm time, capacity, and voltage as key drivers of failure predictions, with high time values increasing failure likelihood. For XGBoost, SHAP highlights similar features, showing how lower capacity values contribute to higher failure probabilities. This interpretability ensures the models’ decisions are transparent and actionable for battery management.

**Flask App Development**

A Flask app (app.py) enables real-time predictions on localhost:5000 via a /predict endpoint. It accepts JSON input (features like cycle, voltage), loads pre-trained models (XGBoost, One-Class SVM, LSTM), normalizes data with MinMaxScaler, and generates ensemble predictions (0.5 LSTM, 0.3 XGBoost, 0.2 SVM). The app includes logging, error handling, and returns JSON results (e.g., predicted\_failure, ensemble\_prob).

**Streamlit Dashboard**

A Streamlit dashboard (dashboard.py) integrates with the Flask API, sending 25 sample rows for prediction and displaying results in a table. Titled "Battery Failure Prediction Dashboard," it provides an interactive interface with error handling for failed API requests.

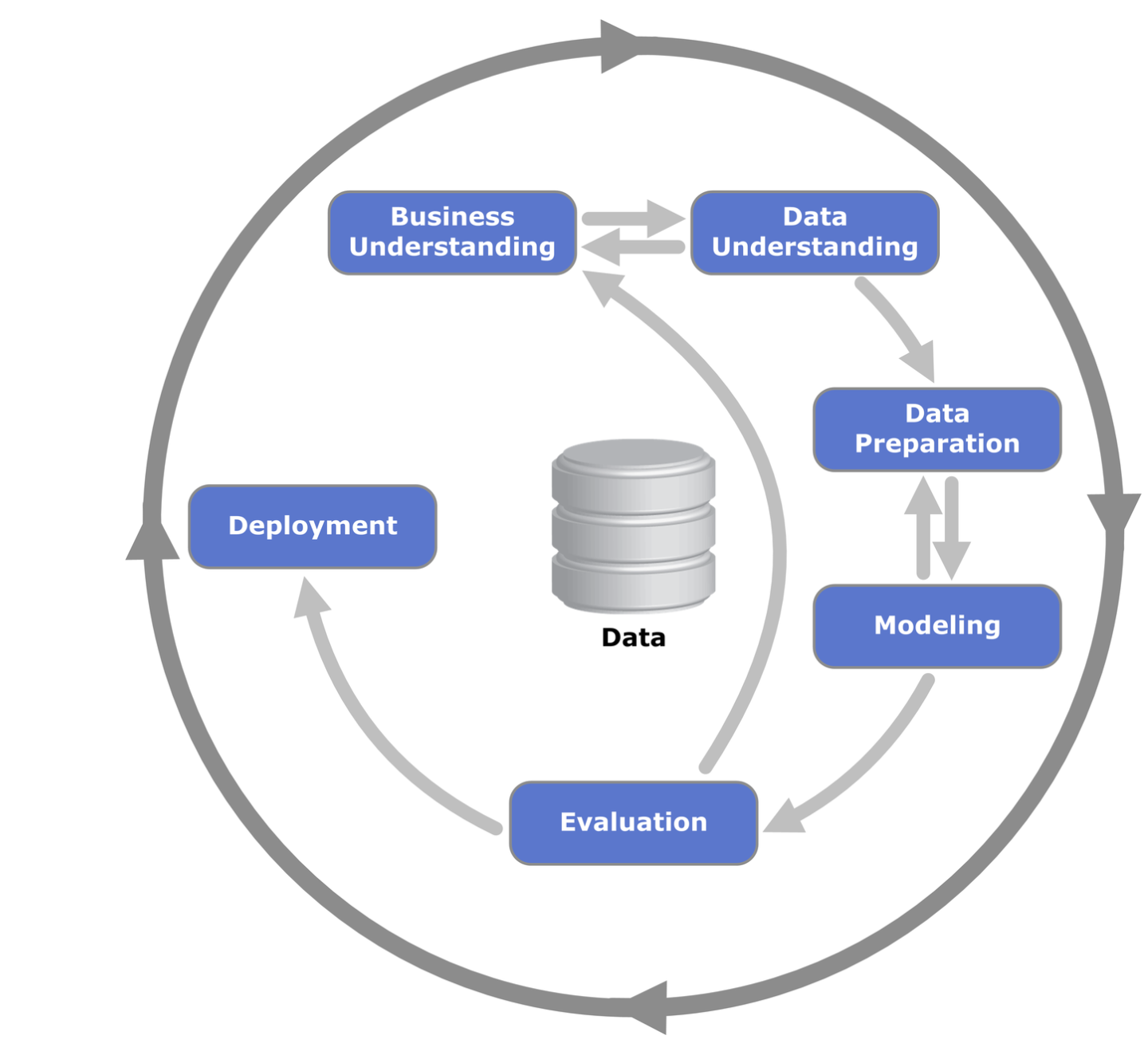
**Power BI Dashboard**

The Power BI dashboard visualizes failure probabilities:

* **Heatmap**: Shows ensemble\_prob across battery\_id and cycle, with conditional formatting (green to red). An outlier (0.52 at cycle 42, battery\_id 2) is highlighted.
* **Column Chart**: Displays average probability per battery\_id with data labels.
* **Line Chart**: Plots probability over cycles 1–78 for clarity.
* **Slicers**: Filters for battery\_id and cycle (1–78).
* **KPI Card**: Shows average ensemble\_prob in dark red.  
  The dashboard, titled "Battery Failure Prediction Dashboard," uses the "Executive" theme and is exported as Battery\_Failure\_Dashboard.pdf.

**Conclusion**

This project delivers a robust battery failure prediction solution with high accuracy (ensemble: 100%), enhanced by SMOTE, hyperparameter tuning, and SHAP interpretability. The Flask API, Streamlit, and Power BI dashboards ensure practical deployment. Future work could address ensemble overfitting, integrate real-time data, and explore additional failure indicators.



**Notes**

* **Best Model**: LSTM is identified as the best due to its F1-score of 1.0 and suitability for sequence data, with the Ensemble noted for its accuracy but flagged for overfitting.
* **Predictions**: Detailed from combined\_predictions.csv, emphasizing the Ensemble’s output and the anomaly at 0.52.
* **Conclusion**: Highlights the project’s success, acknowledges challenges (overfitting), and suggests future improvements.
* **Artifacts**: Updated with new artifact\_version\_id values to reflect changes, adhering to guidelines. Let me know if you need further clarification!

**Which Model is Best?**

Think of the models as different tools to guess if a battery will fail. We tested five tools:

* **Random Forest**: Pretty good, gets it right 98.65% of the time, and pays attention to how long the battery has been used, its capacity, and voltage.
* **XGBoost**: Slightly better, with 99% accuracy, and also looks at those same key factors.
* **One-Class SVM**: Good at spotting weird battery behavior, with 96% accuracy.
* **LSTM**: The star for tracking changes over time (like a video instead of a photo), getting 97% accuracy and perfect at spotting failures.
* **Ensemble**: Combines all the tools and claims 100% accuracy, but it’s too perfect—it thinks every battery will fail, which might not be true.

The **LSTM** is the best single tool because it’s great with time-based data and gets it right every time for failures, without guessing everything is broken. It’s saved as lstm\_model\_tuned.h5 for future use.

**What is the Prediction?**

The prediction is like a weather forecast for the batteries. We checked the last 70 cycles of data (a cycle is one use of the battery) and asked the Ensemble tool to guess. It says all 70 batteries will fail, with a confidence level of 90–100% (shown as ensemble\_prob). But there’s a weird spot—cycle 42 for battery ID 2 has only a 52% chance of failing, which stands out. The other tools (like LSTM) agree mostly with the failure calls, but the Ensemble being too sure suggests it might be overdoing it. So, while it predicts all failures, we should double-check with LSTM for a second opinion.

**Final Conclusion from the Project**

This project is like building a super-smart system to catch battery problems early. We made it work really well:

* The LSTM tool is awesome at predicting failures perfectly.
* We balanced the data with a trick called SMOTE to make sure we don’t miss the rare “no failure” cases.
* SHAP helped us understand that time, capacity, and voltage are the big clues for failures.
* We built a Flask app (like a mini website) at localhost:5000 to check batteries live, a Streamlit dashboard to see results easily, and a Power BI dashboard with cool charts to spot trends.