Equipment Fault Detection & Prediction

Introduction

This data analysis project was developed from scratch and completed independently by Qin Tian using Streamlit.

The **objective** of this project is to detect or predict which equipment may have potential faults based on sensor data. So it is a project including two problems: A multi-class classification for the equipment and a binary classification for the fault. The application consists of 3 main parts:

1. **Data Overview** - Understanding the dataset structure and key statistics.
2. **Exploratory Data Analysis (EDA)** - Analyzing feature distributions, correlations, and class imbalances, etc.
3. **Results Prediction** - Training and evaluating machine learning models to predict faulty equipment.

*for 2~3 are in the **tab** in the frontend in case you didnt notice*

Generally, it is recommended to follow these steps sequentially to gain a better understanding of the entire workflow. However, since the processed data and trained model are provided, you can directly proceed to model inference if needed.

Environment Setup

Prerequisites

- **My Python Version:** `python==3.12.2`
- **Required Packages:** Install dependencies using:

```
pip install -r requirements.txt
```

Setting Up a Virtual Environment (Optional)

To ensure a clean environment, you may create a virtual environment using Anaconda:

```
conda create --name myenv python=3.12.2
conda activate myenv
pip install -r requirements.txt
```

Deployment

To run the Streamlit application, navigate to the `code` directory and execute:

```
streamlit run app.py
```

Main Components

1. Data Overview

The "Data Overview" section is the starting point of my analysis in the Equipment Fault Detection & Prediction project, designed to establish a clear understanding of the dataset before proceeding to deeper exploration or modeling. My objective here was to systematically assess the dataset's structure, quantify its key characteristics, and identify any immediate red flags that could influence preprocessing or model design. This dataset can be downloaded from Kaggle.

Step 1: Loading and Inspecting the Dataset

I began by loading the dataset (`equipment_anomaly_data.csv`), which contains 7,672 rows and 7 columns: `temperature` , `pressure` , `vibration` , `humidity` , `equipment` , `location` , and `faulty` . My first instinct was to verify its integrity—ensuring no missing values or unexpected data types that could derail analysis. Using Pandas' `.info()` , I confirmed that all 7,672 entries were non-null across all columns, meaning no imputation would be necessary. The data types aligned with expectations: `float64` for the five numerical columns (`temperature` , `pressure` , `vibration` , `humidity` , `faulty`) and `object` for the two categorical columns (`equipment` , `location`). This check gave me confidence that the dataset was complete and structurally sound, allowing me to focus on understanding its contents rather than fixing gaps.

Step 2: Understanding Features and Their Context

Next, I mapped out what each column represents to connect the data to the project's goals: multi-class classification of equipment type and binary classification of fault status. Here's how I reasoned through each feature based on the dataset description:

- **Temperature (°C)**: A sensor reading likely tied to equipment operating conditions. High values could indicate overheating, a potential fault trigger.
- **Pressure (bar)**: Another operational metric, where extreme pressures might stress equipment and correlate with faults.
- **Vibration (normalized units)**: A measure of mechanical stability. Excessive or unusual vibration could signal wear or misalignment, making it a prime candidate for fault detection.
- **Humidity (%)**: An environmental factor that might affect equipment performance, especially in humid climates.
- **Equipment**: Categorical labels (e.g., Turbine, Compressor, Pump) representing different machinery types. I assumed each type might have unique fault patterns, critical for the multi-class task.
- **Location**: City names (e.g., Atlanta, Chicago) indicating geographic placement. I hypothesized that location could proxy for environmental differences (e.g., humidity, temperature) affecting fault rates.
- **Faulty**: A binary target (0 = not faulty, 1 = faulty) with a mean of approximately 0.1 (from later stats), suggesting a 10% fault rate and an imbalanced dataset.

This contextual mapping helped me frame the data as a mix of sensor-driven and categorical inputs, setting expectations for how they might interact in EDA and modeling.

Step 3: Quantifying Characteristics with Summary Statistics

To get a numerical handle on the data, I used `.describe()` to compute summary statistics, focusing on the numerical features. This step was essential to establish baselines and spot anomalies. Here's what I found and how I interpreted it:

- **Temperature**: Mean of 70.92°C, ranging from a plausible low to a high of 149.69°C. The maximum seemed extreme compared to the 75th percentile (not shown here but checked separately as 77.57), suggesting potential outliers or rare operating conditions.
- **Pressure**: Mean of 35.74 bar, with a range from 3.62 to 79.89. The spread (std: 10.38 from full stats) appeared reasonable for industrial equipment, but I flagged it for distribution analysis in EDA.
- **Vibration**: Mean of 1.61, but a minimum of -0.43 was striking. Since vibration is a normalized measure and typically non-negative, this negative value suggested either a data error or a

normalization artifact—something to investigate further.

- **Humidity**: Mean of 50.02%, with a tight range (10.22 to 89.98) and median close to the mean, hinting at a near-normal distribution, which could simplify modeling assumptions.
- **Faulty**: As a binary variable, its mean of ~0.1 (computed separately) confirmed a 10% fault rate, signaling an imbalance that could challenge the binary classification task.

These numbers gave me a concrete starting point but also raised questions: Are the high `temperature` and negative `vibration` values meaningful anomalies or noise? Does the fault imbalance reflect real-world rarity or a sampling bias? I earmarked these for EDA to resolve.

Logic Summary

1. **Integrity Check**: Confirmed no missing values or type issues, ensuring a clean slate.
2. **Feature Contextualization**: Linked columns to the dual prediction tasks, hypothesizing their roles.
3. **Statistical Baseline**: Quantified ranges and flagged possible anomalies (e.g., negative `vibration`, high `temperature`).
4. **Visualization Choices**: Used Streamlit to present data intuitively, prioritizing key metrics for quick reference.

2. Exploratory Data Analysis (EDA)

The "Exploratory Data Analysis (EDA)" section is where I dove deeper into the dataset to uncover patterns, relationships, and anomalies beyond the surface-level statistics provided in the Data Overview. My goal was to move from a basic understanding to actionable insights that would inform preprocessing, feature engineering, and modeling decisions. Given that this project tackles both multi-class equipment classification and binary fault prediction, I approached EDA with a dual lens: exploring feature distributions, correlations, and class balances while considering their implications for both tasks.

Step 1: Setting the Stage with Geographic Context

I started by examining the `location` feature, suspecting it might reveal environmental or operational differences affecting equipment performance. Rather than treating it as a simple categorical variable, I mapped each city to latitude and longitude coordinates. I aggregated the data by location, calculating mean values and used the count of observations as a proxy for sample size. In Streamlit, I visualized this on the map, where point sizes reflect observation counts and tooltips display aggregated metrics. The map showed five distinct locations across the U.S., with no immediate

geographic clustering of faults, but it prompted me to consider whether location-specific trends (e.g., humidity in Houston) might emerge in later analyses.

Step 2: Confirming Data Completeness

Before diving into feature relationships, I double-checked for missing values, as they could distort distributions or correlations. Using a bar chart of `data.isnull().sum()`, I confirmed zero missing values across all columns. This wasn't surprising given the Data Overview, but visualizing it reinforced my confidence in the dataset's integrity. It also meant I could skip imputation and focus on the data's inherent properties, a small but critical validation that saved time downstream.

Step 3: Exploring Feature Relationships with Correlation

Next, I investigated how numerical features (`temperature`, `pressure`, `vibration`, `humidity`, `faulty`) interact, as strong correlations could signal redundancy or key predictors. I computed a correlation matrix and visualized it as a heatmap. The results were telling: most correlations were weak (close to 0), except for moderate relationships between `faulty` and `vibration` (positive), and a slight link between `temperature` and `pressure`. This suggested `vibration` might be a critical driver of faults—aligning with my hypothesis that mechanical instability could indicate issues—while other features appeared relatively independent. The lack of strong multicollinearity was a relief, as it reduced the risk of feature redundancy in modeling.

Step 4: Visualizing Feature Interactions with Scatter Plots

To probe these relationships further, I created scatter plots for all pairs of numerical features. Displayed in a three-column layout in Streamlit, these plots showed no clear linear patterns or clusters, reinforcing the correlation matrix's findings. This lack of obvious structure wasn't a dead end—it suggested that simple linear models might struggle, pushing me toward tree-based models like LightGBM that can capture non-linear interactions. It also underscored the need to examine distributions and group differences rather than relying solely on pairwise relationships.

Step 5: Analyzing Feature Distributions

Understanding how features are distributed is crucial for preprocessing and modeling assumptions, so I plotted histograms with KDE overlays for `temperature`, `pressure`, `vibration`, and `humidity`, colored by `faulty`. The results were insightful:

- **Humidity** approximated a normal distribution (mean ≈ median ≈ 50), suggesting it might not need transformation.
- **Temperature** and **Pressure** showed mild skewness, with faulty instances slightly more frequent at temperature extremes—a clue I flagged for fault analysis.

- **Vibration** exhibited a pronounced right skew (long tail), with a minimum of -0.43 from earlier stats. This skewness, combined with negative values, screamed for attention—negative vibration seemed implausible for a normalized metric, likely indicating a data error.

I noted that a log transformation could normalize `vibration`'s distribution, and clipping negative values might be necessary. These histograms gave me a clear visual cue for preprocessing adjustments.

Step 6: Assessing Class Balance

Given the dual classification tasks, I examined the distributions of `equipment`, `location`, and `faulty` using pie charts in a three-column Streamlit layout:

- **Equipment**: Counts for Turbine, Compressor, and Pump were roughly balanced (each ~33%), a good sign for multi-class prediction—no single class dominated.
- **Location**: The five cities were also evenly distributed (~20% each), suggesting no geographic sampling bias.
- **Faulty**: A stark imbalance emerged—only ~10% of instances were faulty (1), consistent with the mean from Data Overview. This screamed "class imbalance" and immediately raised a red flag for the binary task.

I visualized these with interactive pie charts. The `faulty` skew suggested I'd need balancing techniques like SMOTE or class weighting in modeling to avoid bias toward the majority class.

Step 7: Group-Wise Fault Analysis

To dig deeper into fault patterns, I calculated fault rates by `location` and `equipment`, visualizing them as horizontal bar charts:

- **Fault Rate by Location**: Rates ranged slightly (e.g., Atlanta lowest, Houston highest), but differences were minor (~0.08 to 0.12). This suggested location wasn't a strong fault driver, though I kept it in mind for environmental influence via `humidity`.
- **Fault Rate by Equipment**: Turbine, Compressor, and Pump had similar rates (~0.1), indicating no equipment type was inherently fault-prone in this dataset.

These findings, while not dramatic, ruled out strong group-specific effects, shifting my focus to sensor data as the primary fault signal.

Step 8: Distribution by Groups with KDE Plots

I then plotted KDE curves for each numerical feature, split by `equipment` and `location`, to spot group-specific trends:

- **By Equipment**: Distributions were largely overlapping, with slight variations—e.g., Pump had a flatter `temperature` peak, suggesting more spread. `Vibration` curves aligned closely, reinforcing its consistency across types.
- **By Location**: `Humidity` showed more variation (e.g., peaks differed slightly), likely reflecting climatic differences, while other features were similar.

These plots confirmed that group differences were subtle, pushing me to rely on raw sensor values rather than group labels for prediction power.

Step 9: Synthesizing Insights

I summarized my findings in a Streamlit container, listing actionable insights:

1. **No Missing Values**: Data is complete, simplifying preprocessing.
2. **Outliers in Vibration**: Negative values need clipping.
3. **Low Correlation**: `Vibration` is the key fault predictor; others are independent.
4. **Skewness**: `Vibration` requires a log transformation.
5. **Fault Imbalance**: ~10% faulty instances demand balancing.
6. **Balanced Groups**: `Equipment` and `location` distributions are even.
7. **Similar Fault Rates**: No strong group effects.

These insights directly fed into preprocessing (e.g., clipping, SMOTE) and modeling (e.g., focusing on `vibration`), as shown in the app's "Preprocess the data" button.

Logical Summary

1. **Contextualize**: Mapped locations to explore spatial patterns.
2. **Validate**: Confirmed no missing data.
3. **Relate**: Checked correlations and scatter plots for feature interactions.
4. **Distribute**: Analyzed feature shapes and fault overlaps.
5. **Balance**: Quantified class distributions.
6. **Group**: Investigated fault rates and feature splits by categories.
7. **Conclude**: Drew preprocessing and modeling implications.

Even with a "straightforward" dataset, I didn't assume simplicity—I probed for hidden issues (e.g., negative `vibration`) and tested hypotheses (e.g., location effects), ensuring no stone was unturned.

3. Results Prediction

The "Results Prediction" section is where I transitioned from understanding the data to building and evaluating predictive models for the dual tasks of equipment type classification (multi-class) and fault detection (binary). My objective was to leverage insights from the Data Overview and EDA to train effective models, assess their performance, and enable real-time predictions for new data.

Step 1: Model Selection

Given the dataset's characteristics—numerical sensor data, categorical labels, and a notable fault imbalance—I opted for LightGBM (LGBM) as the primary model for both tasks. Here's why:

- **Tree-Based Strength**: EDA showed weak linear correlations but potential non-linear patterns (e.g., `vibration` 's skew), making tree-based models like LGBM ideal over simpler options like logistic regression.
- **Scalability**: With 7,672 rows and a moderate feature set, LGBM's speed and ability to handle categorical variables (after encoding) suited the project's scope, leaving room for future expansion.
- **Multi-Class Support**: LGBM natively handles multi-class problems, perfect for equipment prediction (Turbine, Compressor, Pump).

Step 2: Data Preparation and Feature Engineering

Building on EDA, I used the preprocessed dataset (`equipment_anomaly_data.csv`) from the `preprocess_data` function, which included:

- **Features**: `temperature` , `pressure` , `humidity` , `vibration` , `vibration_log` , `latitude` , `longitude` . The `vibration_log` feature, added due to EDA's skewness insight, aimed to enhance model sensitivity to vibration patterns.
- **Targets**: `le_equipment` (encoded labels for equipment) and `faulty` (binary fault status).
- **Special Handling for Faulty**: For the fault model, I predicted `le_equipment` using the equipment model, one-hot encoded it, and appended it to the feature set. This tested whether equipment type aids fault prediction, despite EDA suggesting minimal correlation.

I split the data into 80% training and 20% test sets with `train_test_split` (stratified for `faulty` implicitly via random seed), ensuring consistent evaluation. Streamlit containers displayed these splits, fostering transparency in how data was fed to the models.

Step 3: Model Training with Hyperparameter Tuning

To optimize performance, I implemented GridSearchCV for both tasks, defining parameter grids based on LGBM's key levers:

- **Equipment Model**: Searched over `learning_rate` (0.01–0.1), `num_leaves` (15–63), `max_depth` (-1 to 10), `min_child_samples` (10–30), and `n_estimators` (100–500), with `objective='multiclass'` and `num_class=3`. This broad range balanced exploration with computational feasibility.
- **Fault Model**: Similar grid but with `objective='binary'`, omitting `num_class`.

Training was triggered via a "View Model Results" button in Streamlit, with spinners (`st.spinner`) indicating progress. If models existed (`lgbm.pkl`), I reused them to save time, reflecting a practical deployment mindset. Results were saved to `./models/equipment/` and `./models/faulty/`, ensuring reproducibility.

Step 4: Model Evaluation and Insights

Post-training, I evaluated both models using accuracy, classification reports, confusion matrices, and feature importance, visualized in Streamlit:

- **Equipment Prediction**:
  - **Accuracy**: 0.57—modest, suggesting the multi-class task is challenging.
  - **Classification Report**: Precision, recall, and F1-scores hovered around 0.55–0.60 across classes (Compressor: 0.57, Pump: 0.54, Turbine: 0.59). The confusion matrix showed overlap (e.g., 231 Compressors misclassified as Turbines), indicating feature similarity across equipment types.
  - **Feature Importance**: `temperature` (4206), `pressure` (4203), and `humidity` (4036) dominated, while `vibration` (2174) and `vibration_log` (1604) lagged. This surprised me—EDA highlighted `vibration`, yet sensor conditions seemed more distinctive here.
  - **Insight**: The moderate accuracy suggested either insufficient feature separation or a need for more data/engineering (e.g., interaction terms), though time constraints limited exploration.
- **Fault Prediction**:
  - **Accuracy**: 0.99—near-perfect, a stark contrast to equipment results.
  - **Classification Report**: Precision and recall were 0.98–0.99 for both classes, with an F1-score of 0.99. The confusion matrix confirmed minimal errors (16 false positives, 21 false negatives out of 2762).
  - **ROC-AUC**: 0.9989, visualized with an ROC curve, reinforcing the model's discriminative power.

- **Feature Importance**: `temperature` (3008), `pressure` (2912), `humidity` (2788), and `vibration` (2468) led, with `vibration_log` (2047) contributing, while one-hot encoded equipment features (e.g., `equipment_predict_Pump` : 128) had negligible impact.
- **Insight**: The high performance suggested fault patterns were clear in sensor data, and SMOTE (from preprocessing) effectively balanced the 10% fault rate. The low equipment feature importance validated EDA's weak `equipment` - `faulty` correlation.

Step 5: Prediction Interface for New Data

I built a user-friendly prediction interface in Streamlit under "Prediction for New Data":

- **Inputs**: Text fields for `temperature` (0–150), `pressure` (0–80), `humidity` (10–90), `vibration` (-1–5), and a dropdown for `location` . Defaults approximated data means (e.g., 70 for temperature).
- **Process**: On clicking "Predict," I preprocessed inputs (clipping `vibration` , adding `vibration_log` , scaling), predicted equipment type, appended its one-hot encoding, and then predicted fault probability.
- **Output**: Displayed as, e.g., "The equipment may be 95% faulty. You may want to check the Turbine." The fault probability (`y_prob`) provided nuance beyond binary classification.

This interface, backed by `make_prediction` , showcased real-time applicability, aligning with the project's predictive maintenance goal.

Logical Summary

1. **Model Choice**: Selected LGBM for its fit to data traits (non-linear, imbalanced).
2. **Data Prep**: Leveraged EDA-driven preprocessing and tested equipment's fault relevance.
3. **Training**: Tuned via GridSearchCV for robustness, reusing models for efficiency.
4. **Evaluation**: Used comprehensive metrics to diagnose performance gaps.
5. **Prediction**: Enabled practical inference with a clear pipeline.

Shortcomings and Future Considerations

This project, while functional and effective for the simplified dataset provided, is a proof-of-concept rather than a production-ready solution. The dataset—comprising 7,672 rows with straightforward sensor readings and a binary fault label—is relatively clean and simple, yielding high fault prediction accuracy (0.99) but modest equipment classification performance (0.57). However, reflecting on

Venti's real-world needs, where a vehicle might trigger multiple alert types with varying severity and complexity, I recognize that this project is a simplified stepping stone.

1. Scalability with More Equipment

Shortcoming: The current model assumes three equipment types (Turbine, Compressor, Pump), achieving moderate accuracy (0.57) with LightGBM. If the number of equipment types grows significantly (e.g., dozens of vehicle alert types), the model's architecture might struggle with increased class overlap and computational demands.
Consideration: To scale effectively, I'd explore hierarchical classification, grouping similar equipment or alerts into super-categories before fine-grained prediction. Alternatively, clustering-based pre-grouping (e.g., using K-means on sensor data) could reduce dimensionality and improve class separation. This would require validating cluster coherence against alert patterns.

2. Handling Unknown or Partially Labeled Faults

Shortcoming: My project assumes fully labeled faults (0 or 1), ideal for supervised learning. In production, like Venti's vehicle alerts, labels might be missing (e.g., undiagnosed spikes) or partial, breaking this assumption.

Consideration: I'd adapt with:

- **No Labels**: Use Isolation Forest for fast anomaly detection, isolating outliers (e.g., high `vibration`) with a tunable `contamination` (e.g., 0.1). If needed, One-Class SVM could refine normal boundaries.
- **Partial Labels**: Combine a small labeled subset (e.g., 500 samples) with self-training—predict pseudo-labels on unlabeled data and retrain iteratively, aided by Z-score to flag outliers.
- **Complex Cases**: Test deeplearning methods like Autoencoders to catch subtle faults via reconstruction errors, scaling up if Isolation Forest misses nuances.

3. Hierarchical Machine Structures

Shortcoming: The project treats equipment as independent units, but Venti's vehicles likely involve hierarchical structures—e.g., a vehicle with multiple components, each triggering distinct alerts. My flat model doesn't account for this complexity.
Consideration: A multi-instance learning (MIL) approach could model vehicles as "bags" of components, predicting alerts at both component and vehicle levels. For instance, if a vehicle's engine (Component A) triggers Type A alerts 60 times, MIL could weigh this against other components' signals.

4. Time-Series Data Integration

Shortcoming: My data is static, capturing single-point sensor readings, whereas Venti's alerts likely evolve over time (e.g., alert frequency or trends). This limits the project's ability to detect temporal patterns critical in real-world monitoring.

Consideration: Transitioning to sequence-based models like LSTMs, Transformers, or Temporal Convolutional Networks (TCNs) could capture alert dynamics—e.g., a rising `vibration` trend preceding a Type A alert. I'd need to reformat the data into time windows.

5. Addressing Overfitting & Underfitting

Shortcoming: The fault model's near-perfect accuracy (0.99) raises overfitting concerns, especially on this small, clean dataset, while the equipment model's 0.57 accuracy suggests underfitting due to limited feature discrimination. I didn't tune beyond GridSearchCV or test regularization.

Consideration: Techniques like dropout, early stopping, or data augmentation (e.g., synthetic sensor noise) could mitigate overfitting, while ensemble methods (e.g., stacking LGBM with RF) might boost equipment prediction robustness. Time constraints prevented these experiments, leaving the single-model approach as a demo limitation.

6. Impact of Equipment Prediction on Fault Prediction

Observation: EDA and feature importance showed weak correlation between `equipment` and `faulty` —fault rates were similar across types, and one-hot encoded equipment features contributed little (e.g., Pump: 128 importance vs. `vibration` : 2468). Adding predicted equipment barely altered fault accuracy (0.99), suggesting it's redundant here. Location (`latitude` , `longitude`) had slightly more influence, possibly via environmental factors like `humidity` .

Concern: In a complex dataset like Venti's, where alert types might tie to specific components (e.g., engine vs. brakes), this independence may not hold. If equipment type impacts alert likelihood, ignoring it could degrade performance.

Consideration: For complex cases, I'd test shared feature extraction (e.g., a neural net backbone for both tasks) or multi-task learning to capture interdependencies. Alternatively, alert-specific models could isolate type effects, but this requires richer data I didn't have.

7. Limited Validation of Feature Engineering and Techniques

Shortcoming: I applied `vibration_log` and SMOTE based on EDA, but didn't validate their impact (e.g., comparing accuracy with/without SMOTE) or explore alternatives like PCA or interaction terms. Time pressure forced me to stick with LGBM without testing RF or deep learning.

Consideration: A/B testing preprocessing steps (e.g., SMOTE vs. class weights) and model variants (e.g., CNNs for feature extraction) would quantify their value.

8. Untapped Dimensionality Reduction and Selection

Shortcoming: I retained all features without PCA or advanced selection, despite low correlations suggesting potential redundancy (e.g., `temperature` and `pressure`). This risks inefficiency in a larger dataset.
Consideration: PCA or recursive feature elimination could streamline inputs, especially if Venti's data balloons to hundreds of sensors. I skipped this to prioritize prediction, but it's a scalability concern.

9. EDA Depth vs. Data Simplicity

Shortcoming: I conducted extensive EDA (maps, KDEs, correlations), but the dataset's straightforwardness—balanced equipment, simple faults—yielded few surprises. This effort felt disproportionate to the insights gained.
Consideration: In a noisier dataset with subtle alert patterns, this EDA rigor would shine, uncovering trends my current data lacks. It's a strength misaligned with this demo's simplicity.

10. Dataset Quality and Real-World Efficacy

Shortcoming: In this project, I assumed the simulated dataset (`equipment_anomaly_data.csv`) mirrors real-world conditions, with its clean sensor readings and binary fault labels driving high fault prediction accuracy (0.99). However, I didn't rigorously assess its quality—e.g., noise levels, representativeness, or alignment with actual equipment failures. In a production setting like Venti's might be diluted by noise or lack clear significance, this assumption could lead to overconfidence. Moreover, Venti may not even have a pre-existing labeled dataset, requiring manual annotation from scratch—an untested challenge in my workflow.

Consideration: To evaluate dataset fitness, I'd calculate metrics like signal-to-noise ratio (e.g., comparing sensor variance to fault signal strength) and perform label consistency checks (e.g., cross-validating fault labels against operational logs). For Venti, correlating model predictions with real maintenance outcomes—e.g., did a predicted Type A alert lead to a repair?—would quantify real-world impact. However, these steps assume an available dataset. If starting from raw, unlabeled vehicle data, manual annotation becomes critical, raising two key questions: *How much data should be labeled, and which features should be prioritized?*

- **Annotation Scale**: Determining "how much" to label depends on balancing cost, time, and model performance. For a baseline, I'd start with a small sample—say, 500–1,000 vehicle

observations (akin to 10–20% of my 7,672-row dataset)—and train an initial LightGBM model, measuring accuracy or AUC. Iterative active learning could then guide further labeling: prioritize uncertain cases (e.g., near decision boundaries) to maximize information gain. In Venti's case, if a vehicle triggers Type A alerts 60 times vs. Type B's 10, I'd ensure enough samples capture both frequent and rare alerts. Statistical power analysis (e.g., based on expected alert prevalence) could refine this, but I'd need pilot data to estimate effect sizes—unavailable here due to the simulated dataset's simplicity.

- **Feature Selection**: Choosing "which features" to annotate hinges on their predictive value and feasibility. I'd hypothesize key signals based on domain parallels: `temperature`, `pressure`, `vibration`, and `humidity` worked in my project, so analogs like engine temperature, tire pressure, vibration frequency, or environmental humidity might apply to vehicles. To judge relevance, I'd run an initial unsupervised analysis (e.g., PCA or correlation on raw sensor logs) to identify high-variance or correlated features, then consult domain experts (e.g., mechanics) to confirm their link to alerts. For example, if vibration spikes precede Type A alerts, it's a keeper; if humidity shows no pattern, it's expendable.
- **Adapting to No Dataset**: If Venti has no labeled data, I'd propose a bootstrapping approach: collect raw sensor streams, apply anomaly detection (e.g., isolation forests) to flag potential alerts, and manually label a subset for supervised training. This hybrid method leverages unlabeled data while building a labeled corpus.

11. Code Structure Limitations

Shortcoming: Time constraints left the code's functional partitioning less modular than ideal, which hampers maintainability.
Consideration: Refactoring into separate modules (data prep, modeling, UI) would enhance clarity and extensibility, a priority for production code I couldn't address.

Conclusion

My project simplifies Venti's challenge—where multiple alert types, varying frequencies, and less obvious signals demand nuanced detection—into a binary fault task. In practice, manually spotting a vehicle needing "Type A focus" from 60 triggers is feasible, but automating this across thousands of vehicles with mixed alerts requires more.

Beyond the listed considerations, I'd prioritize multi-label classification (predicting all alert types simultaneously), anomaly detection for subtle signals, and integration with live data streams. This

demo proves the basics; scaling it to Venti’s needs means tackling these complexities head-on.

Feel free to discuss! 🤗