ChatGPT

Data and Preprocessing

The provided dataset consists of three linked tables (Process data, Laboratory data, and Normalization factors) and 25 high-frequency time-series files (one per product configuration) [1] [2]. The **Process dataset** (1,005×35) contains engineered features from the manufacturing runs (e.g. average press speed, compression force statistics, waste rates) and the key quality attributes (drug release, impurities) [1]. The **Laboratory dataset** (1,005×55) includes raw material and in-process measurements (API assay, moisture, intermediate tablet hardness, etc.) [3]. The **Normalization dataset** (25×3) provides batch-size scaling factors for cross-batch comparability [4] [5]. Each time-series file (labeled 1.csv–25.csv) contains all batches for a given product code, with ~2–20 hour runs sampled every 10 s [6].

Integration steps: Load all data (CSV with ";" separator [7]) and merge by the batch identifier. Specifically, join the Process and Laboratory tables on `batch` [8]. Then merge the normalization factors by product code, so that each batch record includes the appropriate scale factors [5]. Next, align the time-series data: for each batch, extract the segment from the corresponding time-series file (matching on product code and batch ID) and trim to the production window (2–20 h as given by the run) [6] [8]. Synchronize all relevant sensors (recorded every 10 s) and create a unified time-indexed record per batch.

Handling missing and downtime: The documentation notes that *process sensor data are fully complete* (no missing values) [9], but laboratory variables have a few systematic gaps (0.2–0.9% missing, tied to specific API batches) [10]. We will impute these (e.g. batch-mode or model-based imputation) or drop affected samples if negligible. In the time series, machine stoppages may appear as zeros or flatlines (e.g. `tbl_speed=0`). We should detect idle periods (e.g. speed or produced count = 0) and exclude or mask them, since they do not reflect normal production dynamics. Short gaps can be interpolated or forward-filled, but prolonged downtime segments should be removed from training to avoid biasing the models. We will also apply the batch-size normalization factors to any count-based or time-dependent metrics (such as waste, produced counts) [11], ensuring sensor values are comparable across different batch sizes.

Summary of preprocessing steps:
- Load Process, Laboratory, Normalization, and Time-Series files [12].
- Merge Process + Lab on `batch` [8], then add normalization by product code [5].
- For each batch, extract the time-series window and map sensors (by code and batch).
- Normalize relevant signals (apply batch-size factors to waste, production counts) [11].
- Clean missing/loss data: impute rare lab gaps; remove or mask downtime in sensor streams.

Sensor Selection for Forecasting

Not all 16 recorded sensors need to be forecasted; we should focus on those most predictive of final quality. Key process sensors include **compression force** (`main_comp`), **fill depth** (`tbl_fill`), **press speed** (`tbl_speed`), and variability measures like **SREL** (std. dev. of force) [13]. Production monitors (`produced`, `waste`) and **ejection force** (tablet ejection, indicating sticking) are also relevant [14]. To identify the most predictive, we will use a combination of domain knowledge and data-driven analysis:

- **Correlation analysis:** Compute correlations (or mutual information) between each sensor's statistical summaries (mean, max, trend) and final quality metrics (drug release, impurities). This highlights sensors whose behaviors correlate with out-of-spec outcomes.
- **Model-based importance:** Fit a preliminary classifier/regressor on the *static Process/Lab data* (or on features extracted from the full time series) to estimate feature importances. For example, a random forest on process summaries can rank sensors by their influence on quality.
- **Expert judgment:** In tablet compression, final quality is strongly affected by compaction and fill variations. High deviation in compression (SREL) or unexpectedly low force can lead to under-compressed tablets. Ejection force anomalies signal formulation issues. These arguments prioritize forecasting `main_comp`, `tbl_fill`, **SREL**, `tbl_speed`, and `ejection` (see Table below).

Sensor	Meaning	Relevance to Quality
`main_comp`	Compression force (kN)	Directly influences tablet density/hardness and dissolution. Large drops can cause defects.
`tbl_fill`	Fill depth (mm)	Determines dosage volume. Deviations can cause under/over-filling and quality failures.
`SREL`	Std. dev. of compression force (%)	Measures force variability. High SREL indicates inconsistent compaction, risking impurity and dissolution issues.
`tbl_speed`	Press speed (tabs/hour)	Affects dwell time and uniformity. Abrupt speed changes can shift product quality.
`ejection`	Ejection force (N)	Indicates tablet sticking or capping. Spikes suggest formulation problems, leading to defects.
`produced`	Cumulative good tablets	Drop in production rate can flag upstream issues. Useful for detecting slow-onset faults.
`waste`	Cumulative reject count	Direct measure of failures. Sudden increases may precede final product defect.

Table: Key process sensors to forecast, based on the dataset's time-series parameters [15] [16] and their impact on tablet quality.

By focusing forecasting on these variables, we capture the dynamics most likely to cause a batch to fail. Other sensors (e.g. `stiffness`, `cyl_main`, `fom`) may be less directly linked to final quality and can be deprioritized or used for ancillary tasks (anomaly detection). Sensor selection will be refined by exploratory analysis (e.g. cross-correlation plots, preliminary feature importance).

Feature Engineering and Selection

Forecasting model features: We will use past time-series values of the chosen sensors as inputs. Specifically, for each time step *t*, the input can be a multivariate window of the last *W* measurements (e.g.

the past 30–60 min) for each selected sensor. This creates a sliding-window dataset:
- Each example: [$\text{sensor}_1(t\text{–}W),\ldots,\text{sensor}_k(t\text{–}1)$] → predict [$\text{sensor}_1(t+H),\ldots,\text{sensor}_k(t+H)$] for horizon *H* (30–60 min ahead).
- We normalize each sensor's data (e.g. z-score or MinMax) per product code to account for scale differences (batch-size effects are partly handled by normalization factors [11]).
- We may also include engineered time features (elapsed time since start, phase indicators) or simple statistics (recent trend, rolling average). However, care is needed: the model should primarily learn from raw sequence patterns. If using deep learning (LSTM/CNN), we may supply only the raw multivariate sequence and let the network infer features.

Classification model features: The classifier's inputs will be derived from the **forecasted sensor values** at the chosen horizon (30–60 min into the future). Rather than raw 360 (60 min×6/min) individual points, we will likely summarize the forecast window into features: for each sensor: mean, max/min, trend (slope), and possibly the final predicted value. For example, "predicted main_comp average over next 60 min" or "difference between predicted force now vs. 60 min later". This compresses the rich time-series output into manageable features. Additionally, we should incorporate static and lower-frequency data: raw material analyses (e.g. API moisture, impurity) and process-level summaries (e.g. total_waste, startup_waste from the Process table) can be included, since they are known at production start. These features help account for non-sensor causes of defects.

Feature selection will use standard techniques. For forecasting, we already pre-select sensors of interest. For classification, we can apply feature ranking or regularization (e.g. L1, tree-based importance) to prune redundant predictors. As recommended in the documentation, we will drop any pairs of features with extremely high correlation (>0.95) [17] to avoid multicollinearity. We will also apply normalization factors to any features involving batch counts or times [11] [17]. In summary, the feature logic is: use raw forecast outputs (and static variables) transformed into summary metrics, carefully scaled and de-correlated, guided by both domain insight and data-driven ranking.

Forecasting Model Design

We will build a *multi-step, multivariate* time-series forecasting model to predict the selected sensor values 30–60 minutes ahead in real time. Key design points:

- **Model type:** Recurrent neural networks (LSTM/GRU) or Temporal Convolutional Networks (TCN) are well-suited for capturing time dependencies. For example, an encoder–decoder LSTM can ingest 60 min of past data (360 points at 10s intervals) and output the next 60 min. Alternatively, a sequence-to-sequence CNN (1D ConvLSTM) could be used. The model should support multiple inputs (e.g. main_comp, tbl_fill, etc.) and produce multi-output forecasts. One practical approach is a single multi-output model predicting all selected sensors simultaneously, leveraging their correlations. Another is separate univariate models per sensor, but this ignores cross-sensor patterns. We lean toward a unified model (with one output vector per time step).

- **Training approach:** We will create a training set by sliding windows over historical batches. Each batch's time series yields many examples (sliding windows offset in time) up to the process end minus horizon. The loss function can be mean squared error (MSE) summed over all sensors and forecast steps. We may also experiment with weighted losses if some sensors are more critical. If batch durations vary (2–20 h [18]), shorter runs produce fewer training samples, but the large number of batches (1,005) should suffice. We will include product code or batch-size as an input (or build separate models per product code) to account for different operating points.

- **Scalability and performance:** For real-time use, the model must infer quickly. Modern LSTMs/TCNs with moderate depth can make a forecast in milliseconds on a CPU or faster on a GPU. We should keep the model size modest (e.g. 2–3 layers, <100k parameters) to ensure low latency. Once trained offline, the model will be deployed as a service (e.g. via TensorFlow Serving or PyTorch TorchServe) in a streaming pipeline. Batch-style retraining can occur periodically (e.g. weekly) to incorporate new data.

- **Validation:** Evaluate forecasting accuracy using a rolling-origin or blocked time-series split (see below). Metrics include RMSE and MAE on held-out batches. We will also monitor if forecast errors correlate with defect cases (i.e. if forecast misses often coincide with actual quality failures), to ensure the classifier step has reliable inputs.

Classification Model Design

The classification model will predict a **binary defect label** for each batch, using the *forecasted* sensor data as inputs. Steps include:

- **Label definition:** We define a batch as “defective” if its final quality metrics violate specifications. For example, if the *Drug Release Average* is below spec (e.g. <85%) or *Total Impurities* above spec. (Spec limits should be set by domain experts or by observing the historical distribution [19].) Alternatively, one can train continuous regressors on quality targets and flag predictions outside bounds. For simplicity we use a binary target: 1 = out-of-spec (defect), 0 = within spec.

- **Features:** Use the forecast outputs. For each key sensor, include the predicted mean, max, and trend over the horizon. We may also include the last observed (current) value as a baseline. In addition, include static features such as API assay and moisture from raw materials and any engineered features (e.g. total_waste) that are available before or during the run. All features will be scaled appropriately. The documentation suggests creating composite quality scores or efficiency ratios, which we can mimic (e.g. yield = produced/(produced+waste)). Ultimately, the classifier sees a feature vector representing the *expected near-future process state* combined with known inputs.

- **Model type:** Because the feature set is moderate-sized tabular data, gradient-boosted trees (e.g. XGBoost or LightGBM) or random forests are strong candidates. They handle heterogeneous features, require minimal tuning, and give feature importance for interpretability. Alternatively, a simple feed-forward neural network or logistic regression (with regularization) could suffice. Given the critical nature, we prefer an interpretable model. We will monitor precision/recall since defects may be rare.

- **Training:** Simulate real pipeline: for each historical batch, feed its pre-computed time-series up to some point into the forecasting model to generate “predicted” features, then train the classifier on those features against the true quality label. This way, the classifier learns on data that resemble actual forecasts (not on perfect future info). We split data by time/product (see next section) to avoid information leakage. Regularization or feature selection (as above) will control model complexity. We may also consider ensembling models trained on different subsets (e.g. by product code) to improve robustness.

- **Performance:** The classification step is light (inference in sub-millisecond), so real-time speed is easily met. However, emphasis is on accuracy: we evaluate via ROC-AUC, F1-score, and confusion matrix on withheld batches. Because false negatives (missed defects) are costly, we may tune the classifier threshold to favor recall (catch all potential defects), at the expense of some false alarms.

Validation Strategy

To ensure generalizability, we will use **time-based and product-based cross-validation** [20] :

- **Time-based split:** Divide the 2.5-year span into chronological folds. For example, train on batches from 2018–2019, validate on 2020, and test on 2021 (hold-out the most recent period) [20] . This respects temporal ordering and simulates future deployment (the model predicts on newer data than it was trained on).

- **Sub-family (product code) CV:** Since there are 25 product codes (four tablet strengths × various sizes), we will also perform CV by leaving out entire sub-families. For example, train on 20 codes and test on the remaining 5. This checks robustness across formulations [20] . Alternatively, do k-fold where each fold excludes one or a few codes.

- **K-fold for random splits:** While not purely time-ordered, stratified k-fold (grouped by batch) can be used for additional feature selection or hyperparameter tuning, ensuring each batch appears only in one fold.

- **Metrics:** For forecasting, use MAE/RMSE over the forecast horizon on held-out batches. For classification, use confusion matrix, F1, ROC-AUC, and recall at a given false-alarm rate. We will especially monitor results on the hold-out set (latest data) to simulate real performance.

This validation plan follows the dataset's recommendations for *temporal and subfamily* splits [20] and ensures we do not overfit to batch-specific or time-invariant artifacts.

Real-Time Deployment Architecture

The final system will integrate the two models in a real-time data pipeline:

1. **Data Ingestion:** Process sensors stream data into a time-series buffer (e.g. via a streaming platform like Kafka or MQTT). Preprocessing components filter and normalize this stream (removing downtime, applying scale factors).

2. **Forecasting Microservice:** Every defined interval (e.g. each minute or upon receipt of new data), the service takes the latest *W*-long window of sensor data, feeds it to the forecasting model, and outputs predictions for the next 30–60 minutes for each key sensor. This can be implemented as a REST/gRPC microservice or as a function in a real-time analytics engine (e.g. Spark Streaming, Flink).

3. **Feature Computation:** The raw predicted sequences are summarized into features (means, trends, etc.) by a lightweight service. These features are combined with available static data (raw-material specs, current waste) to form the classifier input vector.

4. **Classification Microservice:** The feature vector is then passed to the classification model (deployed similarly). The model returns a defect probability or alert flag.

5. **Integration and Alerting:** If a defect risk is high (above a threshold), the system triggers an alert (to operators or higher-level MES). Alternatively, it logs the prediction for review. All predictions and sensor data are stored for traceability and future model re-training.

6. **Monitoring and Retraining:** System health monitors latency (ensuring each step completes in, say, <500ms) and accuracy (using subsequent lab results to confirm predictions). Periodic retraining jobs update the models with new data (batch mode offline).

Architecture diagram (conceptual):

- Sensors → **Streaming Ingestion** → **Preprocessor** → **Forecast Model** → **Feature Formatter** → **Defect Classifier** → Alerts/Logs.

Both models are versioned and containerized; scaling (with Kubernetes or similar) ensures resilience. This design achieves real-time performance (models run in milliseconds) and can scale to high throughput (hundreds of batches). It aligns with Process Analytical Technology (PAT) goals of real-time quality control by enabling early detection of deviations [20] [18].

Throughout, all decisions (from feature choices to model types) are driven by the dataset's structure and pharma context. For example, we normalized waste by batch size [11] as suggested, and we respect the data's temporal nature in CV [20]. This comprehensive strategy ensures that the forecasting and classification models are both accurate and practical for live deployment in a pharmaceutical manufacturing setting.

Sources: Dataset documentation and technical literature [8] [13] [21] [20]. (All citations refer to the provided data documentation.)

---

[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] Dataset Documentation.pdf
file://file-GT9i8iELEWHSiQv5ZNmXBJ